The Dissertation Committee for Robert John Ascott Certifies that this is the approved version of the following dissertation:

JAVAFLOW: A JAVA DATAFLOW MACHINE

Committee:

Earl E. Swartzlander Jr., Supervisor

Anthony P. Ambler

Derek Chiou

Lizy K. John

Keshav K. Pingali

JAVAFLOW: A JAVA DATAFLOW MACHINE

by

Robert John Ascott, BS; MSEE

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

December, 2014

Dedication

To my wife Virginia, David (1985-2009) and to Colin, Mason, Sofia, Cian, Quentin, and Laurelin whose world of technology will be far different than today.

Acknowledgements

Sincere thanks to Professor Earl Swartzlander Jr. for his encouragement, patience, and continuously positive attitude which made this research enjoyable and productive. Also thanks to the myriad of technical and fellowship groups which maintained and expanded my mental, physical, and spiritual fitness over the past several years.

JAVAFLOW: A JAVA DATAFLOW MACHINE

Robert John Ascott, PhD
The University of Texas at Austin, 2014

Supervisor: Earl E. Swartzlander Jr.

The JavaFlow, a Java DataFlow Machine is a machine design concept implementing a Java Virtual Machine aimed at addressing technology roadmap issues along with the ability to effectively utilize and manage very large numbers of processing cores. Specific design challenges addressed include: design complexity through a common set of repeatable structures; low power by featuring unused circuits and ability to power off sections of the chip; clock propagation and wire limits by using locality to bring data to processing elements and a Globally Asynchronous Locally Synchronous (GALS) design; and reliability by allowing portions of the design to be bypassed in case of failures.

A Data Flow Architecture is used with multiple heterogeneous networks to connect processing elements capable of executing a single Java ByteCode instruction. Whole methods are cached in this DataFlow fabric, and the networks plus distributed intelligence are used for their management and execution. A mesh network is used for the DataFlow transfers; two ordered networks are used for management and control flow mapping; and multiple high speed rings are used to access the storage subsystem and a controlling General Purpose Processor (GPP). Analysis of benchmarks demonstrates the potential for this design concept. The design process was initiated by analyzing SPEC JVM benchmarks which identified a small number methods contributing to a significant percentage of the overall ByteCode operations. Additional analysis established static instruction mixes to prioritize the types of processing elements used in the DataFlow Fabric.

The overall objective of the machine is to provide multi-threading performance for Java Methods deployed to this DataFlow fabric. With advances in technology it is envisioned that from 1,000 to 10,000 cores/instructions could be deployed and managed using this structure. This size of DataFlow fabric would allow all the key methods from the SPEC benchmarks to be resident.

A baseline configuration is defined with a compressed dataflow structure and then compared to multiple configurations of instruction assignments and clock relationships. Using a series of methods from the SPEC benchmark running independently, IPC (Instructions per Cycle) performance of the sparsely populated heterogeneous structure is 40% of the baseline. The average ratio of instructions to required nodes is 3.5. Innovative solutions to the loading and management of Java methods along with the translation from control flow to DataFlow structure are demonstrated.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

Section 1.1 - Computing Challenges

JavaFlow, a Java DataFlow Machine employs high levels of both innovation and invention to address a series of computing challenges being faced by those attempting to implement modern computing platforms. The challenges addressed in this project include:

- Design complexity and associated design resource requirements
- Power consumption and cooling constraints
- Limited performance of wires across the chip
- Reliability in large system on chip implementations
- Management of software in multi-core structures

Innovation is achieved through the integration of a series of currently trending technologies:

- Virtually unlimited number of circuits on a chip
- Advanced on chip networks
- Dataflow architectures
- Java Virtual Machine Specification

Inventions are applied to the concepts of a DataFlow machine which allow the dynamic loading of a Java ByteCode method and execution of this procedural language structure within the dataflow framework. The two key inventions in this design are:

- The use of a self-organizing system to dynamically load and perform address resolution of Java methods.
- Additions to a traditional DataFlow machine to allow whole procedural Java methods to be resident in a DataFlow fabric and execute with high

performance and power efficiency exploiting higher levels of locality than in other computing structures.

In his book "The Smart Swarm" [1] Peter Miller described behavior of leaderless groups achieving great results. Examples included bees, geese, ants all of whose populations achieve a series of common goals without centralized leadership. These techniques are applied to the distribution of a Java ByteCode method into a DataFlow fabric in order to prepare for computation. The assignment of specific instructions to specific locations is not made centrally, but rather developed as the program is deployed throughout the network of DataFlow processing elements.

DataFlow machines exploit locality for working variables, but traditionally have had difficulty implementing loops and registers for communications across basic blocks. Dynamic DataFlow machines have used extensive hardware to both handle this looping and achieve high levels of parallel execution of loops. The JavaFlow machine demonstrates low cost structures to effectively deploy local registers locally to the processing elements and to implement both forward and loopback branches without adding traditional Dataflow switch/merge/predication functions.

The machine utilizes a minimalist approach to the design of the processing elements in the DataFlow fabric so that the maximum number of nodes can be realized on a chip. By utilizing the DataFlow fabric itself to load and resolve producer/consumer addresses, constraints on the instruction set encoding can be relieved with data expansion in the processing elements. One example of this is fan out from one producer to a number of consumer nodes.

Section 1.2 - Hardware Background

Price and performance have been the two factors used to measure the effectiveness of computing systems since they were first created. While these two factors have at times been combined into a single metric, the largest change in the measurement of effectiveness has come in the components that make up these factors. In the early days of computing the system cost was primarily associated with financial cost of the materials used to implement the hardware of the computer. Similarly, the performance was initially measured as the frequency of the clocking circuit that drove the hardware machine. Even in the early years the cost of development was a consideration in the overall price of the system, and system clock speeds needed adjustments to account for instruction set differences.

In later years the components of these factors have changed dramatically. Cost measurements now start with area of a semiconductor chip and include the cost of the design. However; the increasing cost of complexity, verification, and testing plays an increasing factor in the overall ability to implement a processor design. The power dissipation from a traditional design on a modern semiconductor chip may yield a design that is either not feasible due to cooling or requiring significant cost in cooling technology.

With the differences in performance of components such as memory, circuitry, and I/O; clock speed-up on traditional micro-architectures has not led to comparable system performance increases. This is where advances in architecture and micro-architecture have combined with technology speed-up to improve system performance. Specifically, areas such as cache memory, pipelining, and parallel execution of some instructions have provided most of the system performance gains over the past 10 years.

Details of these various hardware alternatives are described in Chapter 3: Related Work.

SECTION 1.3 - SOFTWARE BACKGROUND

The original programming model was coding in either machine or assembly language to achieve the optimal performance for a specific application. The very high development costs of this machine level programming strategy has led to a plethora of high level languages over the past 50 years. All of these languages were aimed at increasing the productivity of the programmer while not sacrificing too much performance versus the hand coded assembly/machine language.

The Java language became popular in the late 1990's and its popularity remains today. Tiobe BV [2] has attempted to measure 'popularity' of programming languages by applying metrics to web search results. While this does not necessarily demonstrate the business usage of these languages, it does offer one measure of 'popularity.' Figure 1 demonstrates this 'popularity' measurement with Java shown as the top graph. The second most 'popular' language is C with the recent cluster of C++, Visual Basic, and PHP showing in third place.

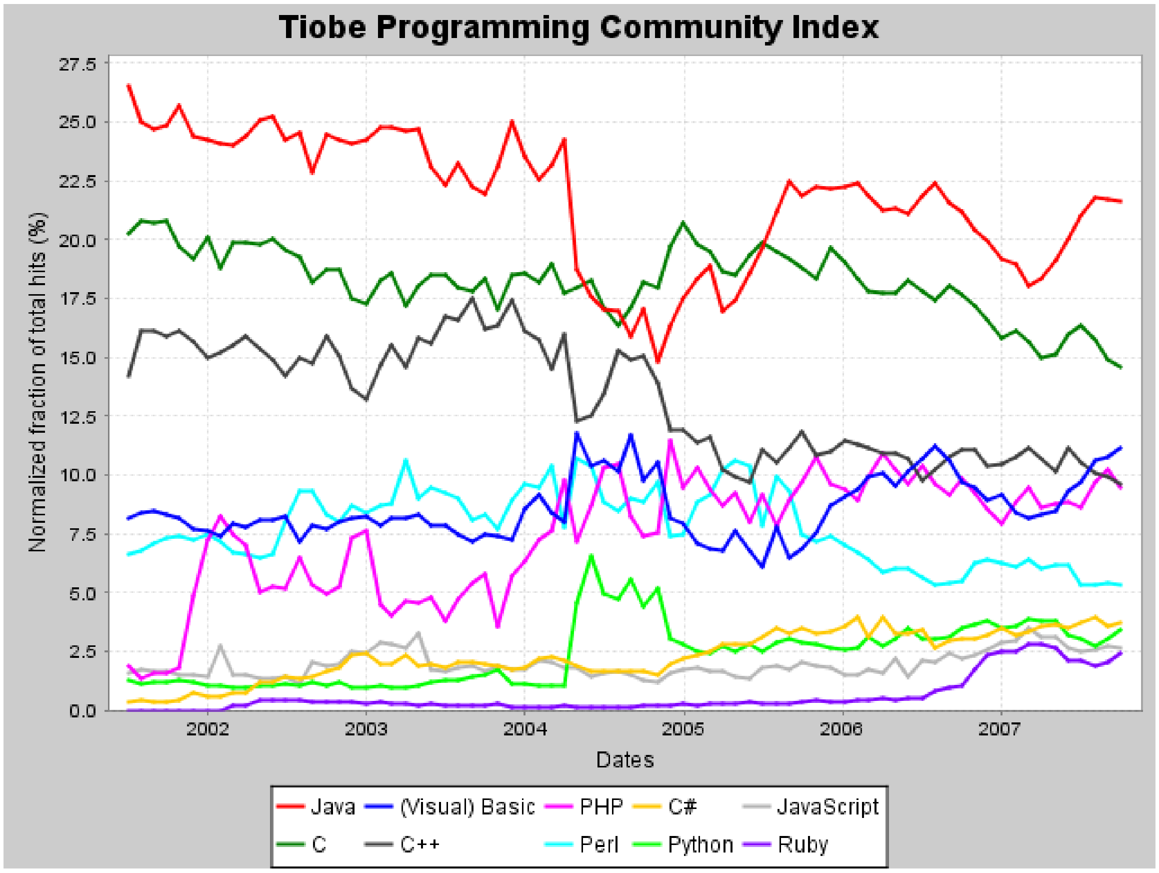
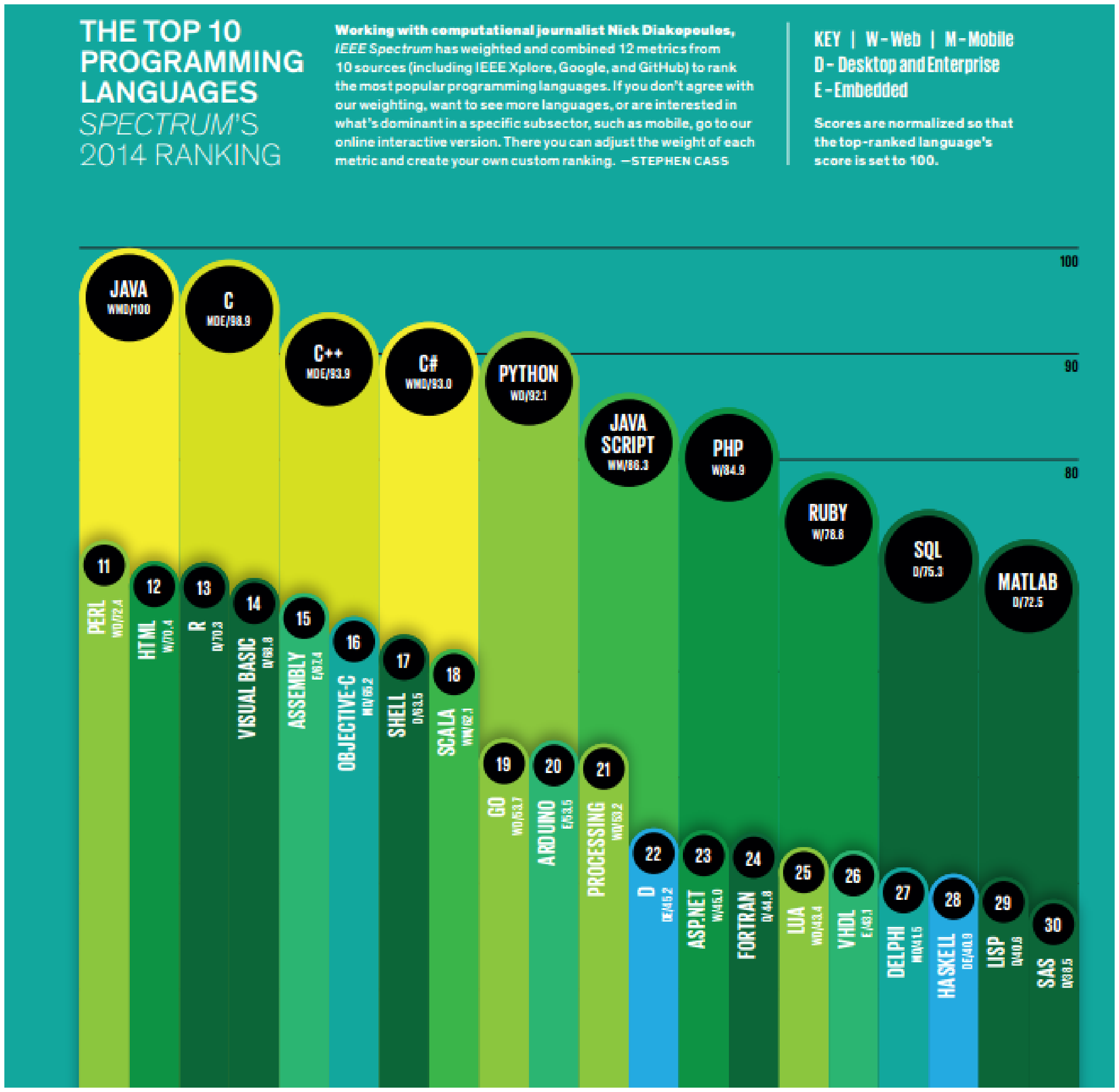


Figure 1 Tiobe Programming Community Index [2]

A more recent review of programming languages was published in July, 2014 [3] and again, Java is the achieved the highest ranking as shown in Figure 2. This latest ranking used metrics from 10 sources including IEEE, Xplore, GitHub, and Google to attempt to quantify the popularity of languages.


THE TOP 10 PROGRAMMING LANGUAGES SPECTRUM'S 2014 RANKING

Working with computational journalist Nick Diakopoulos, *IEEE Spectrum* has weighted and combined 12 metrics from 10 sources (including IEEE Xplore, Google, and GitHub) to rank the most popular programming languages. If you don't agree with our weighting, want to see more languages, or are interested in what's dominant in a specific subsector, such as mobile, go to our online interactive version. There you can adjust the weight of each metric and create your own custom ranking. —STEPHEN CASS

KEY | W-Web | M-Mobile
D-Desktop and Enterprise
E-Embedded

Scores are normalized so that the top-ranked language's score is set to 100.

Figure 2 IEEE Spectrum 2014 Language Ranking [3]

Java is one of the later languages aimed at standardizing the complex world of programming. Despite advocates of alternative languages, the Java language has achieved some level of standardization. This is due the definition of the "Java Virtual Machine" (JVM) [4, 5] which has become ubiquitous on all modern computing platforms. There are multiple computing scenarios where Java is utilized:

One scenario is the internet and a computing model which distributes software over the network to a remote computer/browser for a specific computing assignment. The JVM represents an intermediate, machine level architecture which has been implemented on almost all current hardware platforms for the execution of downloaded programs over the network. Key attributes of this JVM are that programs cannot negatively affect the computing platform, and a series of processes are in place to insure the integrity of the distribution of these programs. With these two factors in place, the scope of computing platforms for which the JVM can be effective expands to include cell phones and the ever increasing set of small computing machines which previously may have been considered 'hard coded' by their original designers.

At the other end of the application spectrum, Java is a major component of Enterprise Computing Systems. The programmer productivity from levels of abstraction and software reuse is key to this popularity. The JavaFlow machine with its focus on managing large numbers of cores is more likely targeted towards this application area than the more cost sensitive personal device marketplace.

The JVM is a stack based architecture whose instructions are called *ByteCodes*. Each instruction contains a single byte operation code and a variable number of operands. Due to the stack nature of this architecture, instruction level parallelism has been difficult to obtain.

Chapter 2: JavaFlow Problem Statement

Section 2.1 - Summary

This section offers a brief statement of the problem addressed in this research and a summary of the solution. The following section describes the related work, and the last section presents details of the proposed solution. Chapter 1: Introduction introduced a series of problems being addressed by computer architects.

The specific problem addressed in this research is focused on the general problems identified in Chapter 1 in the context of a Java Virtual Machine executing ByteCode instructions. The specific direction of this solution is the definition and analysis of a Java Dataflow machine. This machine combines the basic concepts of DataFlow machines along with recent developments in this field to implement a hardware ByteCode execution subsystem.

The overall goal is to use a minimalist approach to implement a JVM while implementing complete methods inside a DataFlow Fabric. This will address performance issues of traditional ByteCode approaches and exploit the advantages of a DataFlow machine which can capitalize on the data locality of the Java ByteCode program.

An additional goal of this machine is the automation of the loading and management of the DataFlow Fabric to relieve the operational systems from these tasks. With the number of cores anticipated to grow to between 1,000 and 10,000, the importance of this automation task will expand in the future.

Section 2.2 - Included Topics

The research effort includes the analysis of comparable hardware and software solutions and the analysis of SPEC benchmarks to gain understanding of the dynamic and static behavior of key Java Methods. The DataFlow machine is described and the ability to load/unload/manage/execute ByteCode instructions is demonstrated. A baseline machine configuration is identified and comparisons to a series of DataFlow configurations is made against the baseline. Instructions per Cycle, DataFlow node utilization, and method DataFlow parameters are measured.

Discussion is provided for handling of all instructions and special cases such as exceptions are addressed. The management of the Java Method Area and Heap is discussed and ways of performing garbage collection is addressed.

SECTION 2.3 - EXCLUDED TOPICS

As described in the Proposal for this Dissertation, a series of items are not included in the scope of this research:

- Detailed logic design of DataFlow Nodes
- Semiconductor analysis of power management techniques
- Specific bus widths and implementations
- Complete JVM implementation and hence the ability to completely execute benchmarks on this proposed machine.
    - Detailed implementation of Heap management including Garbage Collection
    - ClassLoader implementations
- Memory subsystem details
- General purpose processor details

Chapter 3: Related Work

SECTION 3.1 - RELATED WORK SUMMARY

Other than the original paper [6], there is no specific previous work on a Java DataFlow Machine. As JAVA has become increasingly popular over a wide range of applications, there are existing architecture structures that successfully implement the JVM. Just as JavaFlow is a combination of hardware and software, most existing solutions have both components. Most solutions employ a combination of:

- General purpose processors
- Compilation techniques
- Special purpose hardware

In addition selected DataFlow machines and other tiled architectures are described.

SECTION 3.2 - GENERAL PURPOSE PROCESSORS

Most implementations of Java Virtual Machines are done with a general purpose processor using various compilation techniques. Modern processors have evolved through many stages:

- Simple single instruction execution
- Pipelined instruction execution
- SuperScalar (multiple instruction issue)
- Multi-core systems

Each stage of this evolution was aimed at both advancing performance, and only at the last step has minimizing power consumption and design effort become significant concerns Modern processors have mostly stabilized on the x86 or ARM ISA (Instruction Set Architecture) and have focused on technology and micro-architecture enhancements to achieve system performance improvements.

SuperScalar is the term applied to micro-architectures that have multiple instructions being issued simultaneously. Sometimes SuperScalar implies out of order execution and deep pipelines. Another key component of the performance gain is in the area of branch prediction. As stated earlier, cache memories are critical to the overall system performance.

A problem with superscalar implementations [7] is that the control logic necessary to support the out-of-order execution is: complex; power consuming; and can be distributed across the entire chip. The complexity translates into increased development cost with its associated risk of design failure. Wiring is becoming a limiting factor as propagation times across the chip have become an increasing percentage of a processor's clock cycle. An additional observation from the effect of the current design strategy is that less than 10% of a modern processor chip is dedicated to arithmetic execution vs. 90% utilized for complex control functions and storage [7]. This 10% utilization and with global wiring delays not scaling as technology scales [8], future designs are focusing on compartmentalizing functions on the chip and optimizing power consumption.

Current approaches to achieve more performance using current technology include focusing on coarser level parallelism. Examples of this include multiple processing cores on a single chip. A key advantage to this structure is that design costs can be reduced through the re-use of existing SuperScalar designs. A further advantage is that whole cores can be powered off when work load decreases, which could result in significant power savings. The challenge in these systems is that typical desktop applications do not present much parallelism. Extracting parallelism to utilize the increased computing power is another very complicated challenge. Note that for server implementations multiple cores or Sun's Niagara [9] advanced Simultaneous Multi-Threading (SMT) structure represent solutions to highly parallel workloads.

With current multi-core designs at 10-100 cores, JavaFlow's simple Instruction Node should provide at least an order of magnitude increase in the total number of Instruction Nodes available to be applied to an application.

While multi-core designs do provide savings in design complexity due the repetition of a single core design, the challenges are in the effective usage and management of these many cores. Since most applications are not designed with parallelism in mind, the partitioning of a single application into multiple threads running on multiple cores is complex, and has been the subject of much research. In addition to the effective execution on multiple cores, the management of these applications is also complex. JavaFlow attempts to address these challenges by allowing a single application to consume a variable number of cores and for the deployment of the program to these cores to be managed by the DataFlow fabric.

SECTION 3.3 - COMPILATION TECHNIQUES

Current Java virtual machines utilize a combination of compiler techniques. All Java programs are initially compiled to the Java ByteCode architecture which offers a series of security and program distribution advantages over other languages.

The simplest way to implement the JVM is on a general purpose processor that first loads the compiled ByteCode class and then interprets the instructions as required to execute the defined method. Modern systems use heuristics to identify methods which are executed repeatedly. Then advanced compilation technology is employed to optimize the execution. This is called Just-In-Time compilation (JIT). Obviously these compilers are specific to the target machine architecture. Finally several specific JVM hardware implementations have been created with the objective of further improving the cost/performance of Java byte code execution.

Even with a JIT compiler, the interpretation of ByteCodes initially remains a key component of most JVM's. Nicolaescu and Veidenbaum [10] analysis of the SpecJVM98 benchmarks showed the compiler performed at between 1.75 and 13.9 times as fast as a baseline interpreter. One of the performance challenges of interpreters is the use of program switch structures to decode ByteCode instructions and the effect on branch prediction in modern SuperScalar processors. Casey, Ertl, Wien, and Gregg [11] showed that branch prediction using BTB (Branch Target Buffers) mispredict 81% to 98% of the indirect branches in switch dispatch interpreters, and 57%-63% in an alternative structure. Their work attempts to further optimize this performance. The JavaFlow machine with its control flow to data flow translation system described in Chapters 4 and 6 does not utilize branch prediction hardware and therefore optimizes this aspect of the JVM in an alternative manner.

In addition to compiler techniques to execute the Java Virtual Machine, it should be noted that a significant aspect of the overall performance of Java programs comes in the Garbage Collection strategy. Note that one of the many advantages of the Java language is that memory management is not a user function, but rather delegated to the JVM. This function is outside the scope of this dissertation, but is mentioned for completeness.

One example of this Garbage Collection strategy has been demonstrated by Azul, Inc. [12]. Azul originally developed a hardware JVM but recently have focused primarily on the Garbage Collection. In enterprise Java applications, where the overall memory usage is beyond the benchmarks used in this analysis, the effectiveness of the Garbage Collection becomes an increasing factor in overall performance.

SECTION 3.4 - SPECIAL PURPOSE JAVA MACHINES

Traditionally a Java Virtual Machine (JVM) [4] is either interpreted or compiled 'just in time' (JIT) on a general purpose processor. However over the past years, several hardware designs have been proposed to implement the JVM and from these designs several key characteristics of the JavaFlow machine can be found.

In general, these machines are aimed at the low cost/power application space, although some employ special hardware for application specific optimization.

The earliest design was the JEM1 [13] processor developed by Rockwell. Its instruction set was modeled after the JVM ByteCode definitions and used traps to execute more complex ByteCodes.

Another early design was Sun's PicoJava [14]. This machine implemented the Java stack in hardware and introduced the concept of 'instruction folding.' This reduces the number of ByteCode instructions executed by optimizing movements of operands from the local storage to the top of the stack. A configuration diagram of PicoJava is shown in Figure 3

Schoeberl defined the Java Optimized Processor (JOP) [15] and reviewed six additional designs targeted towards embedded systems [16]. The JOP dataflow is shown in Figure 4 [15]. While Java optimization is performed, the machine is a traditional von Neuman architecture.

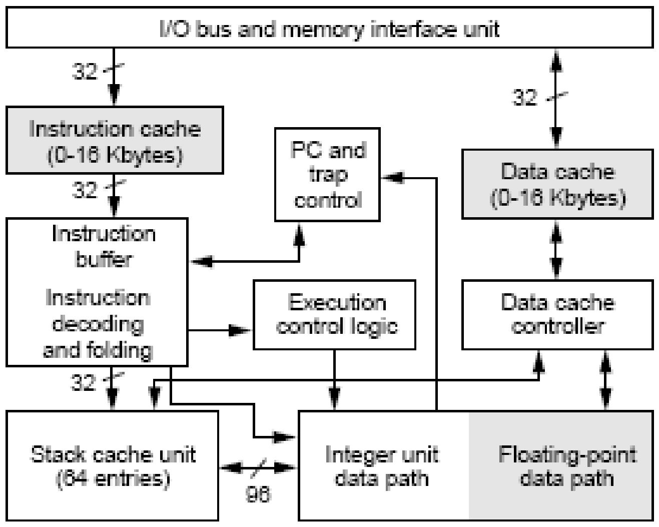


Figure 3 Pico Java Configuration [14]

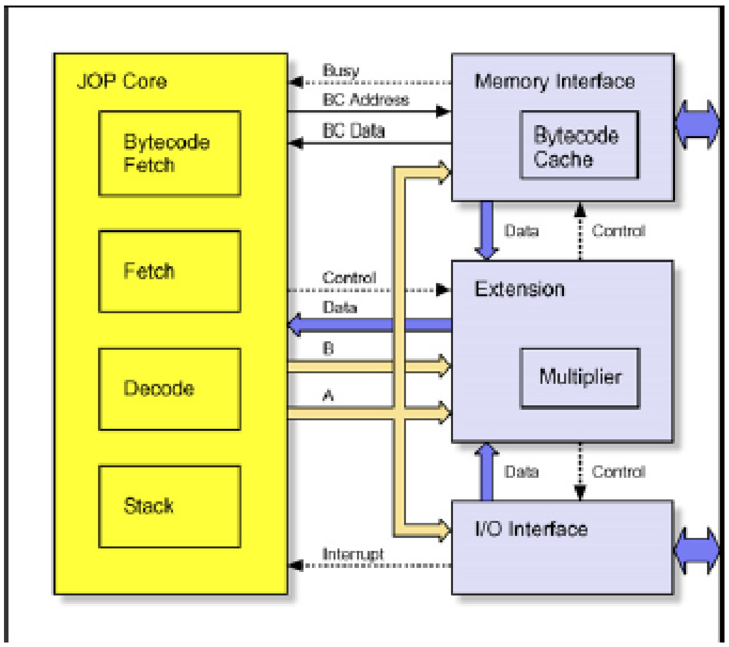


Figure 4 JOP Machine Structure [15]

The Molen FemtoJava Engine [17] expanded on a traditional Java hardware core to perform application specific functions outside of the main Java execution unit.

Radhakrishnan defined further optimizations to the folding algorithm of PicoJava and also defined ‘Hard_Int’ [18, 19] which optimized ByteCode execution by performing folding offline and by combining instructions into hardware macros.

Other efforts continued to seek optimizations on the execution of ByteCodes by eliminating, re-ordering, and translating ByteCode instructions [20-22].

In [23] Wang and Yuen demonstrated instruction level parallelism by using data flow concepts to tag JVM stack operands to realize both folding and out of order execution in a VLIW structure.

In [24] Vijaykrishnam, Ranganathan and Gadekarla extended the focus of the hardware implementation to include support for the object-oriented aspects of the Java programs. In addition to folding techniques implemented elsewhere, this machine used new cache constructs to assist the access of both fields and methods of Java objects.

Although not a specific Java machine, a recent multicore offering is relevant to see the potential for a machine like JavaFlow. Adapteva, Inc. released its Epiphany 64 core processing chip [25] where a general purpose processor is used in conjunction with a tiled fabric of processors. Each node has a full processing capability and the mesh network implements a single memory address space accessible by all cores. An interesting aspect of this design was part of a conversation with the company president which indicated the entire design was completed in a relatively short period of time by a very small design team. This design was released to the Global Foundry’s semiconductor facility and sample parts with 16 or 64 cores are available. This architecture is claimed to scale to 4096 cores which is the magnitude envisioned for the JavaFlow machine.

Another special purpose hardware solution entering the Java space has been defined by the Heterogeneous Systems Architecture Foundation [26]. This system architecture utilizes the combination of a set of general purpose processing cores combined with a

Graphics Processing Unit which contains potentially thousands of cores. The GPU cores execute in a SIMD (Single Instruction, Multiple Data) configuration where an extreme level of parallelism is available for tasks that can be partitioned to exploit this system. The target application for these systems, as contained in their name is the processing of graphic images for high performance display rendering. Recent research has been targeted in exploiting this high level of parallel processing capability towards parallel applications. In addition to the limitations of the SIMD structure, traditional GPU's require the transfer of data between local memories and the shared main memory of the general purpose processing cores.

The HSA architecture overcomes a significant limitation of traditional GPU's by allowing each processor to directly access the large memory space of the general purpose processor cores. This capability combined with the structures supported in the latest release of Java (Java 8 [27]) which explicitly identifies parallelism through Lambda expressions allows performance enhancements of specific parallel applications through the execution on HSA GPU systems. This work is part of the Sumatra Project of the OpenJDK foundation [28]. These systems are aimed explicitly at improving the performance of highly parallel applications rather than the general applications targeted by the JavaFlow machine.

An additional demonstration of hardware enablement can be found in IBM's TrueNorth "neuromorphic chip" [29]. While not aimed at Java processing, it demonstrates the "unlimited number of circuits" referenced in the Introduction. This 2014 chip claims 5.4 billion transistors, 4096 "neurosynaptic" cores, 1 million programmable neurons, and 256 million programmable synapses. They also claim a system of 16 chips. A key aspect of this design is the power density claim of 20mW per square centimeter; or 70 mW for the chip.

SECTION 3.5 - DATAFLOW MACHINES

History and concepts:

In 1975 Dennis [30] proposed a computing architecture as an alternative to the traditional von Neumann model. The term applied to this architecture was a 'DataFlow Machine' because it was structured according to the dataflow graph of the underlying computing algorithm. Conceptually this structure could employ significantly more circuits for arithmetic and logical operations versus control. Practically, however; this was not the focus of early dataflow implementations due to the overall scarcity of circuitry. Modern technology has reopened the door for the implementation of DataFlow machines where much of the complex control circuitry could be turned into additional arithmetic and logical operators with less global control requirements.

There has been recent work in the area of DataFlow machines. Some key problems with the earlier designs, such as memory ordering have been overcome by the modern machines enabling them to be considered for modern general purpose computing problems.

A key reason for an interest in a tiled DataFlow machine versus a SuperScalar alternative is that the DataFlow machines exploits 'data locality' in the program. For example a tiled DataFlow machine can capitalize on the physical relationship between instructions that produce data and those instructions that consume the same data. SuperScalar machines through general purpose register renaming ignore this relationship and consume significant circuitry and power to allow parallelism and out-of-order execution.

In addition to the work of Dennis, another reference to this technology is available from an earlier Thesis and follow-on publication by D. A. Adams at Stanford in 1968 [31-

33]. Adams was focusing on ways to improve parallel processing and proposed the use of the dataflow graph of the program to sequence instructions. Also, Rumbaugh [34] proposed an early DataFlow machine.

These initial machines were all based on the dataflow graph of a program, but differed in number of tokens that could be present on each dataflow graph edge, on whether execution could begin before all tokens were present, and in the definition of the control nodes in the dataflow graph.

The concept of this machine is to move from the traditional von Neumann computer definition where each instruction is executed in an order specified by a program counter to a machine executing operations in an order based on the availability of data. By basing execution on the availability of data, the expectation is increased performance; and with availability of more hardware, increased parallelism.

DataFlow machines execute according to the data flow graph of a program. Each node of the dataflow graph represents an operation in the program, and the arcs represent paths that data travels between operators. In the pure sense of a DataFlow machine, there is no concept of a program counter or program control. Each operator obeys 'dataflow firing rules' which are defined as having all operands present at its input terminals. All DataFlow machines reviewed so far have nodes with 1 or 2 data operands as inputs and 1 or more nodes as outputs. Terminology used in these structures is 'producer-consumer' where each node consumes data, and then produces a result which is subsequently consumed by another node in the dataflow graph. Figure 5 from Dennis [30] shows a dataflow representation of a simple program. Note that links L1 and L2 are initially enabled. L1 and L2 are links that are 'fired' and make copies of data available to downstream consumers. Operation A1 can 'fire' first. When A1 produces its result, A3 can 'fire.' Subsequently A4 can 'fire' which produces the final result 'x.' Finally this result is sent to A2 which can then 'fire' to produce the final result 'y.' This example showed the firing rules and some parallelism that can be achieved. No conditional operations are shown in this example.

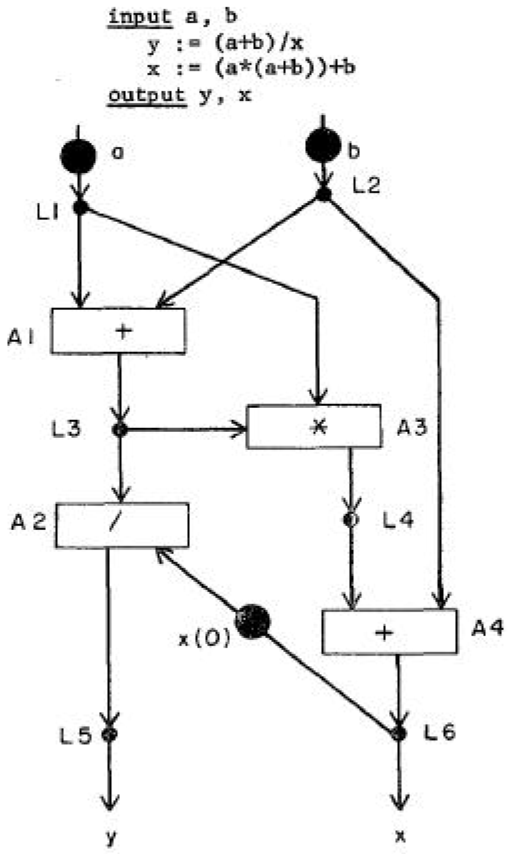


Figure 5 An Elementary DataFlow Program [30]

One of the first challenges of a DataFlow machine is implementation of program control in this data only environment. One of the techniques utilized is predication which translates control dependencies into data dependencies. Different DataFlow designs use different variants to realize this predication. Examples include PHI and inverse-PHI instructions. These have also been called T-Gate, F-Gate, and Switch instructions. The

PHI instruction consumes either of the two operands and passes to the producer side of the node based on a third predication value passed in the dataflow graph. The inverse PHI instruction directs one of the operands to either of two consuming nodes based on a similar Boolean signal. There are many variants on the implementation of predication in DataFlow machines. The WaveScalar machine implements both functions but designates the inverse-PHI function as a Steer instruction [35].

TRIPS uses a different form of predication called “Dataflow Predication” [36] where each instruction has extra bits to identify whether a third predicate operand is required before ‘firing’ can occur.

Beck, Johnson, and Pingali [37] demonstrated the “Access Token” to order memory references in a DataFlow Machine. This structure is used as the MEMORY_TOKEN in the JavaFlow machine, and additional serial tokens are defined in Chapter 6 forming a key aspect of JavaFlow.

Early examples of DataFlow machines were created with technology constraints limiting the number of functional units available to implement the DataFlow graph. Original proposals [38] called for only a single functional unit with the DataFlow 'nodes' saved in an array whose execution order was determined by the DataFlow firing rules. There were several problems with these designs that precluded a practical solution.

First the single functional unit precluded any level of instruction level parallelism, but did allow increased occupancy of the processor. This was consistent with most modern processors in the 1970's where a single functional unit (arithmetic and logical unit) was commonplace. Second, the logic of early DataFlow machines to determine the firing rules consisted of a large associative array to match the availability of an operand from one operation to the requirements of other operands. These arrays were not only expensive but inserted delay into the overall processing pipeline.

Another differentiating characteristic of DataFlow machines is how multiple values of a variable are handled. This is key to loop structures and high levels of parallelism. A Static DataFlow machine allows only one variable to be present on any arc in the control flow graph. While this avoids the problem of multiple values in loop iterations; potential parallelism is lost. The alternative is to add a tag to each token (variable) to distinguish values in different iterations of a loop. This "dynamic tagged token" approach is utilized in the initial MIT machine [38] and also in WaveScalar [35].

One aspect of early DataFlow machines was the challenge/opportunity regarding the ordering of memory operations. With a traditional von Neumann architecture, the program counter implies an order to the memory operations. On the other hand, a DataFlow machine following only DataFlow firing rules might create an unexpected ordering of memory load and stores. The solution to this challenge has been the topic of much research over the years and two solution areas have emerged. The initial focus of DataFlow machines was in the use of Functional Languages [39]. These languages offered parallelism, but were significantly different from traditional languages and they have not seen widespread use among programmers. A characteristic of the memory used in these languages is that after initialization, it can only be written once so that read-after-write data hazards are eliminated.

Recent DataFlow implementations have imposed an ordering on memory operations so that traditional imperative languages can be executed. WaveScalar, TRIPS, and the JavaFlow machine all employ memory ordering which may preclude some parallelism to insure program integrity.

Monsoon, Manchester

Early research and resulting implementations of DataFlow machines focused both functional languages and maximizing the parallelism while deploying a relatively small number of processing units. These implementations were targeted for large scale scientific processing. Each implementation required extensive matching logic for a significant number of tokens which would then be dispatched to the execution unit(s). Each implementation used dynamic tagging of the operands.

The Manchester machine [40] demonstrated parallelism on programs written in the single assignment language SISAL. Tokens are carried in data packets around a pipeline ring structure where a matching unit looks for pairs of tokens that can be sent to an execution unit. The Manchester machine has a microcode controlled pipeline while the Monsoon machine is implemented exclusively in hardware.

The original MIT architecture was called TTDA (Tagged Token DataFlow Architecture) and was followed by the ETS (Explicit Token Store) architecture [38]. This latter architecture formed the basis for the Monsoon implementation. These machines exploited the functional language Id [39]. The TTDA architecture had the same challenges as the Manchester machine where the matching of the dynamically generated tokens was very expensive in both logic and cycle time. The ETS/Monsoon extended the architecture to replace the matching functions with an explicit location where the first argument to a binary instruction is saved and presence bits set to non-empty. Upon the arrival of the second argument, the presence bits for the location are read indicating that the first argument has arrived. Then the first argument is read and the instruction is fired. The presence bits are then reset to empty.

TRIPS

The TRIPS project at the University of Texas [41, 42] is focused on "EDGE" Explicit Data Graph Execution. The project is a combination of a static DataFlow machine with a program counter to maintain control flow. Because of this approach towards the execution of a traditional program structure, this machine is the closest comparison to the JavaFlow design. A system diagram of the TRIPS machine is shown in Figure 6.

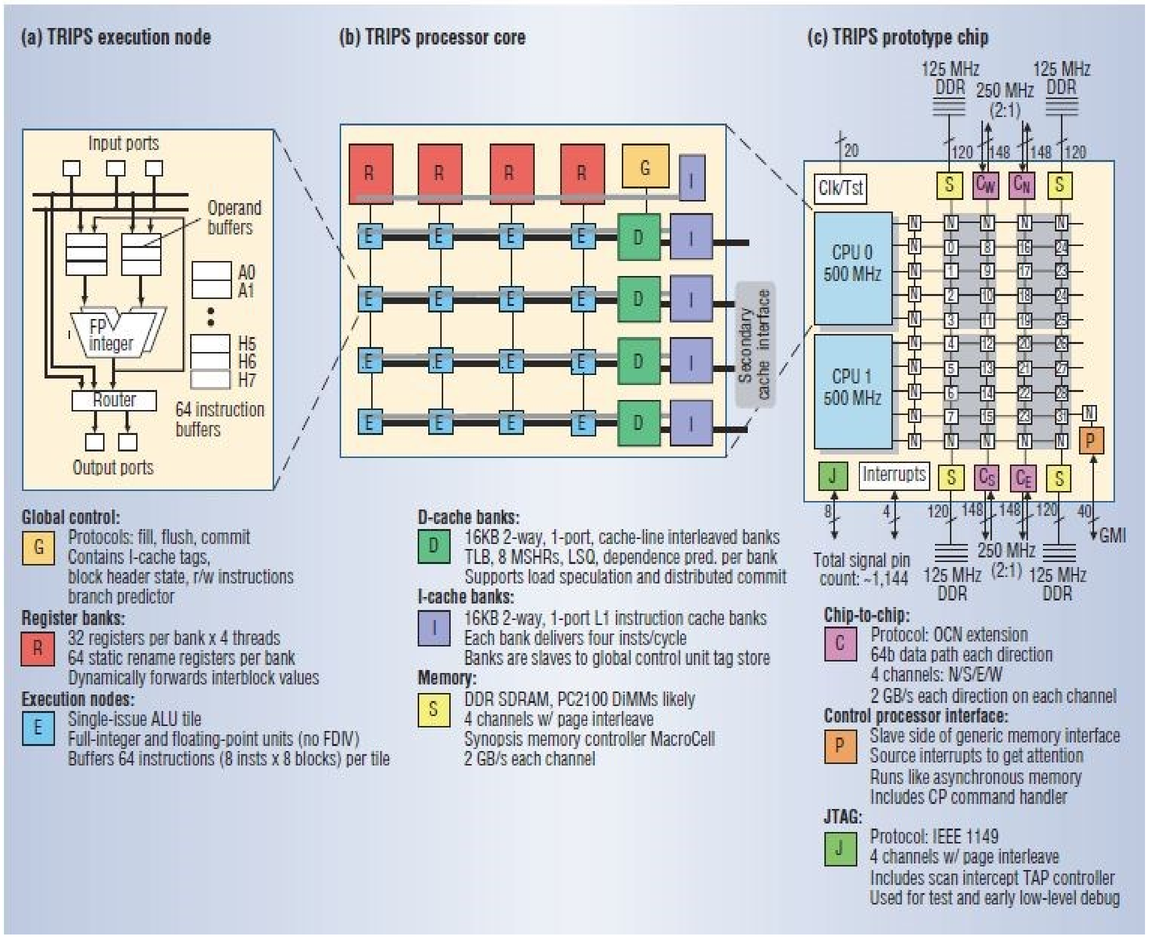

(a) TRIPS execution node
Input ports
Operand buffers
A0
A1
H5
H6
H7
FP integer
Router
64 instruction buffers
Output ports
(b) TRIPS processor core
R
G
I
E
D
Secondary cache interface
(c) TRIPS prototype chip
125 MHz DDR
250 MHz (2:1)
Clk/Tst
CPU 0 500 MHz
CPU 1 500 MHz
S
C_W
C_N
C_S
C_E
N
P
J
Interrupts
GMI
Total signal pin count: ~1,144
Global control:
G Protocols: fill, flush, commit
Contains I-cache tags, block header state, r/w instructions
branch predictor
Register banks:
R 32 registers per bank x 4 threads
64 static rename registers per bank
Dynamically forwards interblock values
Execution nodes:
E Single-issue ALU tile
Full-integer and floating-point units (no FDIV)
Buffers 64 instructions (8 insts x 8 blocks) per tile
D-cache banks:
D 16KB 2-way, 1-port, cache-line interleaved banks
TLB, 8 MSHRs, LSQ, dependence pred. per bank
Supports load speculation and distributed commit
I-cache banks:
I 16KB 2-way, 1-port L1 instruction cache banks
Each bank delivers four insts/cycle
Banks are slaves to global control unit tag store
Memory:
S DDR SDRAM, PC2100 DIMMs likely
4 channels w/ page interleave
Synopsis memory controller MacroCell
2 GB/s each channel
Chip-to-chip:
C Protocol: OCN extension
64b data path each direction
4 channels: N/S/E/W
2 GB/s each direction on each channel
Control processor interface:
P Slave side of generic memory interface
Source interrupts to get attention
Runs like asynchronous memory
Includes CP command handler
JTAG:
J Protocol: IEEE 1149
4 channels w/ page interleave
Includes scan intercept TAP controller
Used for test and early low-level debug

Figure 6 TRIPS Machine Organization [41]

This processor breaks down the control graph of a program into ‘hyperblocks’ which can contain up to 128 instructions. The DataFlow fabric consists of 16 Processing Elements (PEs) and the instructions of the hyperblocks are distributed across the PEs. Each PE contains up to 64 instructions which are addressable as consumer nodes from producer and predicating instructions. This combination shows that up to 1024 instructions can be dispatched or ‘in flight’ during program execution, although at most 16 could be executing in parallel. Each of these processing elements follows the DataFlow firing rules with predication having been included in the compiled code to implement branching. Hyperblocks are defined as blocks of code where no loops are allowed.

The key to the execution of programs is that inter-block communications is handled primarily through the register banks which eases the pressure on memory references. The intra-block data communications is handled with the DataFlow producer-consumer transfers using the edges of the dataflow graph. A further key to execution is the ‘block atomic’ execution of each hyperblock. A hyperblock is a segment of code that executes atomically where there is a single entry point, possibly multiple exit points, and no internal looping. Once a hyperblock is initiated, the system can deterministically know when it is complete independent of the path taken through the block. Each hyperblock is allowed to make 32 register reads and writes and the completion of the block is defined by the completion of all register writes.

A critical factor in the successful implementation of TRIPS is the compiler [43]. Since the hyperblock size is 128 instructions and most program blocks are significantly smaller, the compiler is responsible for loop unrolling to fill the instruction space. Furthermore, the compiler is responsible for address assignment to minimize the distance between producer and consumer nodes and the proper handling of all register reads and writes. A challenge for the instruction set of the TRIPS is that limiting the instruction width to 32 bits allows only 2 possible consumers for the data that is produced at a processing element. This limitation requires the insertion of move or fan-out instructions that takes a single data element and moves it to two additional consumers.

TRIPS implements a memory ordering scheme similar to WaveScalar which insures proper handling of data dependencies. TRIPS utilizes a series of on chip networks optimized to handle the producer-consumer operand transfers; the instruction loading; register read/write; and access to the L2 data cache.

The TRIPS machine was implemented using standard cell 130 nanometer technology. Analysis of the implementation showed mixed results when compared to other SuperScalar machines. [44] The largest challenge in the analysis was to normalize the technology and design resources when attempting to compare this project with commercial processors. Code benchmarks were analyzed compared to an Alpha architecture to measure instructions executed, parallelism, storage accessed, and instructions per cycle using both compiled and hand optimized code.

The conclusion of the analysis was that the EDGE execution structure offered no clear advantage over a traditional architecture nor advantage over a modern technology implementation. Several areas were cited as weakness areas which have been incorporated into the design of the JavaFlow machine. The following issues were reported: [44]

1. The fan-out limitation of two consumer addresses caused 20% of the instruction count to be the special move instructions which was larger than expected.
2. The hyperblock size had issues of both oversized block headers and many NOPs required to fill the block. This consumed storage but not processing. Variable sized blocks and block headers would save significant storage and

resulting instruction fetches. This requirement also place significant pressure on the compiler to maximize the block sizes.

3. Since the blocks did not allow looping, branch prediction was key to optimizing the instruction fetches. This added complexity and miss-predictions added execution time to the results.

In summary, the analysis concluded that TRIPS could sustain 10 IPC showing a three-fold cycle count speedup over an Intel Core 2 process with hand optimized kernels. However with compiled benchmarks the cycle counts were not competitive. While technology improvements may show improved results, the change of ISA to this new structure is unlikely to change desktop systems. However, this architecture may apply to systems where high performance and low power are both required such as mobile and data center applications.

WaveScalar

WaveScalar represents another recent DataFlow architecture and implementation and has solved several problems of historical machines. This processor, developed at the University of Washington [7, 35, 45] exploits advanced semiconductor technology to implement a true DataFlow machine with processing elements across a single chip. Processing Elements (PEs) are combined into domains and clusters to support different routing protocols over increasing distances. This fabric of PEs is called the WaveCache.

The simplest DataFlow machine would allocate a single instruction to a PE, WaveScalar assigns 64 instructions to each PE. The goal in the assignment for producer-consumer instructions to be close to each other to avoid network delays while not in the same PE to allow parallelism. A single cluster WaveScalar implementation contains 32

PEs and can store 2048 instructions. A four cluster machine would contain 128 PEs and 8192 instructions. An example of the WaveScalar configuration is shown in Figure 7.

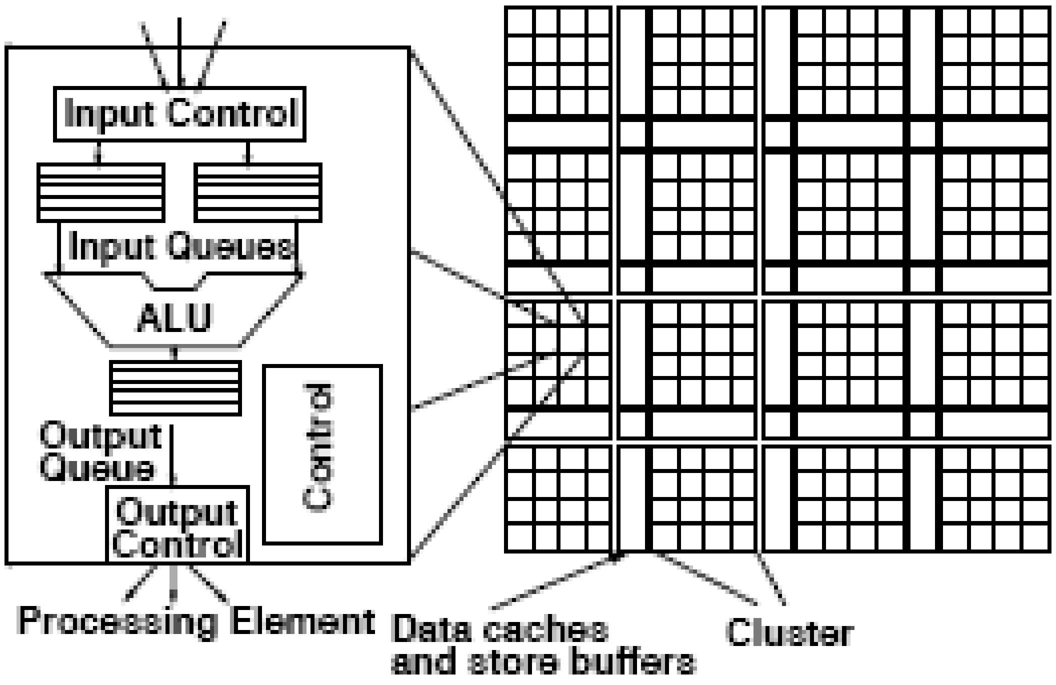


Figure 7 WaveScalar Processing Element [46]

Each Domain consisting of 8 PEs has a section of a banked L1 data cache, store buffers, and a floating point unit. On the periphery of the chip is an L2 cache. Cache coherency is maintained using a MSEI protocol.

The machine was targeted to execute the Alpha instruction set which is then translated to the native DataFlow instruction set. Performance is measured in equivalent Alpha instructions per second.

WaveScalar introduced the concept of 'Waves' which are segments of a program's control flow graph and the ordering of memory operations within waves. This key element allows the WaveScalar machine to execute traditional languages compiled to the Alpha ISA.

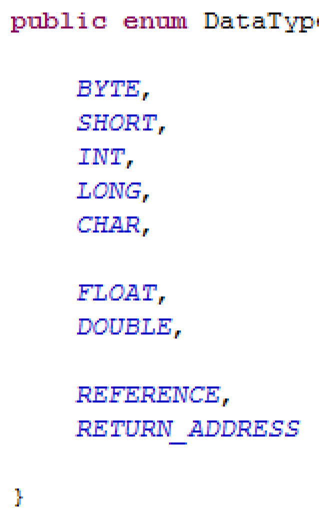
The 'Wave Cache' also supports the transition of unused instructions from the PE's to the memory to make room for required executions. WaveScalar implements a 'Dynamic DataFlow Machine' similar to the original machines previously described. This structure allows multiple instances of loops to be executing concurrently which is key to increases in levels of parallelism. The challenge with this architecture is the increase in design complexity and circuit count to handle the dynamic allocation of machine resources to implement the parallel loop execution.

SECTION 3.6 - JAVA

The JAVA programming language and its associated Java Virtual Machine has some characteristics that invite implementation on a machine like JavaFlow. The Java Virtual Machine is described in an original edition [4] and then updated for Java 8 [47].

While the stack-based architecture has some limitations, there are several characteristics that can be exploited to optimize performance. All local variables and working registers are part of the stack. All accesses to this stack are explicitly addressed by 'ByteCode' instructions. This means that at instruction decode time, all stack addresses are available, and optimizations can be performed.

The Java Language is strongly typed, which means all data is identified according to one of the language types as shown in Figure 8. This removes any ambiguity regarding how to handle specific data elements. The following is a list of specific characteristics of the Java Virtual Machine which key to the success of the JavaFlow machine:



```
public enum DataType {

    BYTE,
    SHORT,
    INT,
    LONG,
    CHAR,

    FLOAT,
    DOUBLE,

    REFERENCE,
    RETURN_ADDRESS

}
```

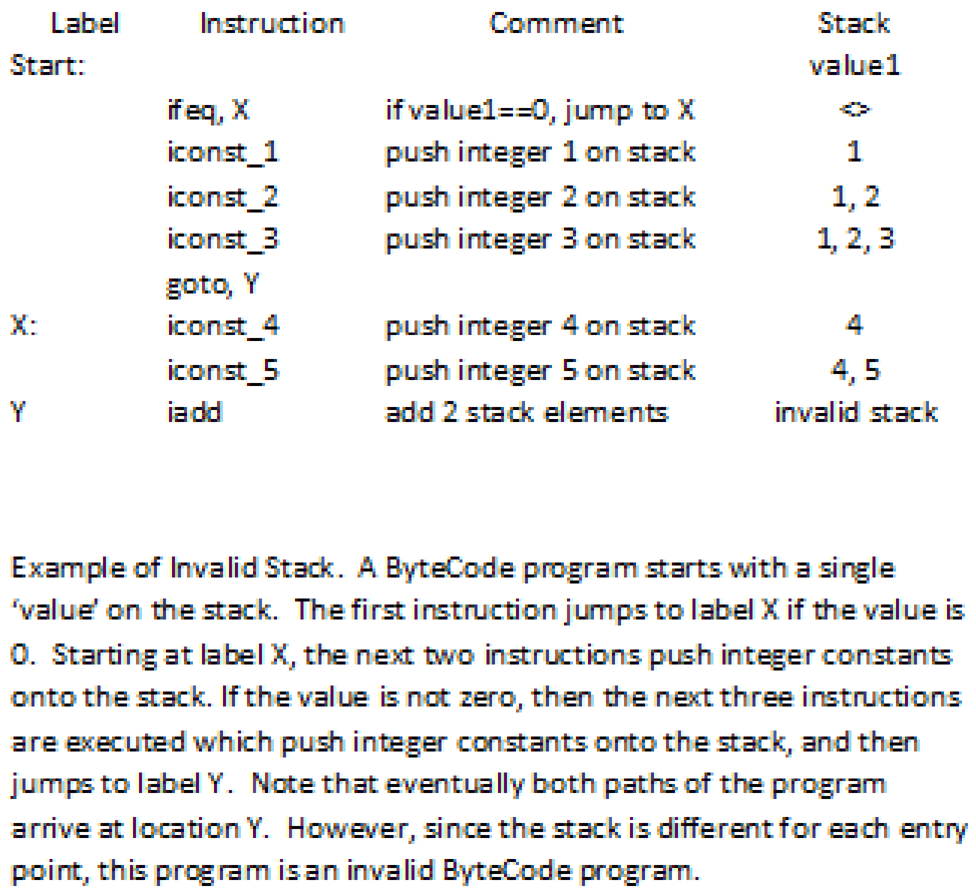
Figure 8 JavaFlow DataTypes

1. The JAVAC compiler provided by Oracle is traditionally used to create the architecturally defined Java Class Files which are eventually loaded into the Java Virtual Machine.
2. Java Byte Code programs have the maximum number of local registers utilized and the maximum number of stack elements defined at compile time. This allows the JavaFlow machine to know if a program would extend beyond the register/stack capacity in the DataFlow Fabric.
3. Although not optimized, the JAVAC compiler utilizes the stack for communications inside a basic block of code and uses local registers for communications between blocks and other methods.
4. Part of the JVM definition is that every instruction must have the same stack configuration from any entry point. An example of this restriction is in the case of a control flow merge. If instructions A and B have instruction C as their next instruction, then the numbers and types of elements of the stack

after the execution of both instructions A and B must be identical. Figure 9 provides an example of this where a ByteCode program starts with a single 'value' on the stack. The example shows only a forward branch, but this situation can become more complicated in the case of back branches or loops.

5. Local Storage addressing is never indirect. All register accesses are specified as absolute values as either part of the opcode or part of the operand. No calculations are allowed which simplifies the interface between the Serial Network and the Mesh Network during Local Register operations.

6. Java programs are organized into Classes and each Class is compiled into a data structure called a ClassFile. Each ClassFile has several components including:
    a. Constant Pool. The Constant Pool is the collection of all constants used by the Class along with definitions/references to all components of the Class such as the Fields and Methods.
    b. Methods. A Method of a Java Class is the actual program consisting of a list of ByteCode instructions. A Java Class may have many Methods and all are contained in the ClassFile.
    c. Additional debugging information and tools to assist in the development and execution of the Class.



Label	Instruction	Comment	Stack
Start:			value1
	ifeq, X	if value1==0, jump to X	<>
	iconst_1	push integer 1 on stack	1
	iconst_2	push integer 2 on stack	1, 2
	iconst_3	push integer 3 on stack	1, 2, 3
	goto, Y		
X:	iconst_4	push integer 4 on stack	4
	iconst_5	push integer 5 on stack	4, 5
Y	iadd	add 2 stack elements	invalid stack

Example of Invalid Stack. A ByteCode program starts with a single 'value' on the stack. The first instruction jumps to label X if the value is 0. Starting at label X, the next two instructions push integer constants onto the stack. If the value is not zero, then the next three instructions are executed which push integer constants onto the stack, and then jumps to label Y. Note that eventually both paths of the program arrive at location Y. However, since the stack is different for each entry point, this program is an invalid ByteCode program.

Figure 9 Invalid Stack Example

Two key elements of the Java language are the use of abstraction via object oriented programming and the use of automatic memory management provided by the Java Virtual Machine. The allocation and management of memory in a Java system is not architected by the Java machine specification, but rather is left up to specific implementation decisions. Venners [5] described several options for managing both the Method Area and the Heap. Tradeoffs in terms of performance and optimized garbage collection are necessary in order to realize an optimized system. The translation process from the symbolic references included in Java Class files to the actual pointers to data is complex and offers both opportunities and challenges to the JavaFlow machine configuration. The affected instructions include memory ‘get,’ ‘put,’ and ‘invokes’ for both static and instance references. The ByteCode instruction definitions [4, 47] call for the operand of these instructions to point to an offset in the Class ‘Constant Pool’ where another pointer into the Class data or the Object instance data is found. The translation from the symbolic reference to this offset can be done by the Linkage process before the method is loaded into the machine or when the data is first accessed, although the details of this process are machine dependent. Once the actual pointer is found, interpreter systems change the opcode by prefacing the code with a ‘_Quick’ modifier to avoid the architected indirection to access the Class or Instance data. This latter approach is the basis for the simulation and performance analysis data reported in Chapter 7. Vijaykrishnan [24] describes this process for both gets and calls in a Java hardware machine which does not have the level of distributed processing capability of JavaFlow.

However; each of the 3 steps to the address resolution process offers potential opportunities for overall performance improvement with associated expense in circuitry. The opportunities involve the utilization of the Instruction Nodes with memory access instructions to resolve these addresses in parallel. The cost is the additional logic required in each node.

- The Class File with symbolic references requires the linking of these references with other Classes that are already loaded or will be loaded. While doing this in the Fabric could offer offload of the General Purpose Processor, the initial judgment is that this function requires visibility to the entire state of the Java Machine and is best left to the GPP. Also, sending the symbolic references to the individual Instruction Nodes would require network traffic and storage in the nodes.

- A more interesting option is to include the architected offset into the Constant Pool as the operand of storage operands and then utilize the Instruction Nodes to resolve these to actual memory pointers by accessing this Class Constant Pool data. While the linking process would still need to resolve all references to other classes, each memory access Instruction Node would be able to find and save the pointer into the Heap or Class area either at load time or memory operation execution time. Note that accesses to static fields are made directly into the Class data area of the Method area, while accesses to objects are first referenced by an object reference variable passed via local registers or the stack.

Figure 10 shows an example of the Constant Pool, Method Area and Heap in a Java memory system. Note that since constants including memory reference information are part of the Constant Pool which is loaded prior to execution, the JavaFlow machine can perform unordered parallel accesses to gather constants for upcoming instruction execution. The Method Area is used to house the actual code in interpreted systems and the Class variables. The Heap is used to house object instantiations of Classes and is clearly subject to Garbage Collection during execution.

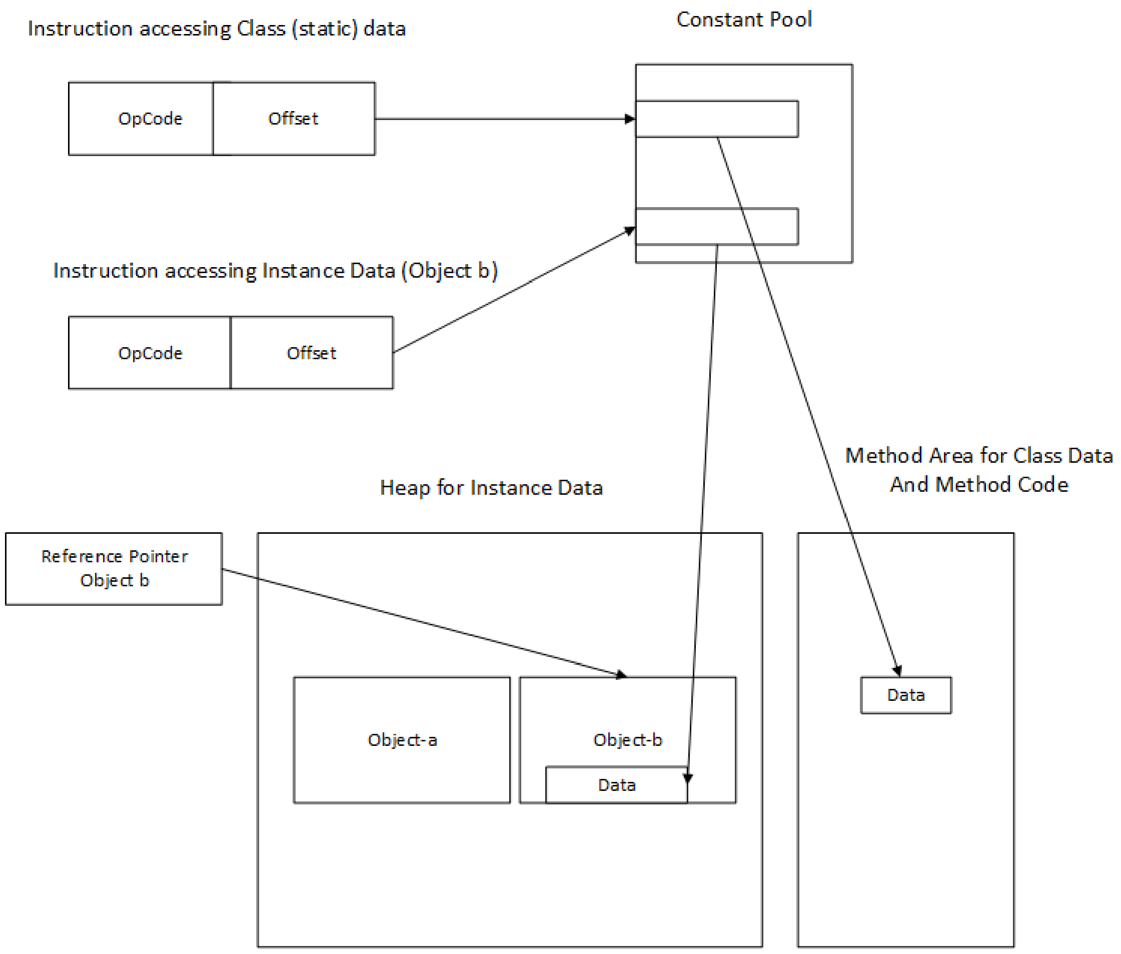


Figure 10 Java Memory Organization

Two example instruction are shown accessing Class and Instance data. For Class data, the instruction contains an offset into the Constant Pool where an offset is found into the Method or Class data area. For instance data, a similar lookup is performed into the Constant Pool, however the offset is with respect to the instance data on the heap which is found by an address reference pointer. Note that to ease access in the current method, any non-static method has its local register 0 containing the reference to the method's instance area on the heap.

Since this research began, Java has realized is version 8 release. Throughout all Java releases, the JVM has remained very constant, and so far this holds for Java 8. The new JVM does provide support for a new type of dynamic method invocation which can be exploited by other languages, the strongly typed Java language still does not allow such structures.

For reference a complete list of the Java ByteCode instructions is provided in Appendix A. The instructions are categorized by group whose processing functions are similar. For each instruction, the contents of the JVM Stack before and after execution is described along with the 'Pop' and 'Push' counts. These counts are the number of stack elements removed and replaced for each instruction.

SECTION 3.7 – SUMMARY

This chapter reviewed a broad spectrum of the technologies available for the execution of Java programs. General Purpose Processors represent a significant area of computer architecture research, and therefore were addressed as they face challenges in technology scaling in the future. Compilation techniques which are the current way Java programs are primarily executed would be limited by the scaling issues facing General Purpose Processors. Special purpose Java machines demonstrated techniques to optimize the execution of ByteCode instruction streams which is key to JavaFlow and its proposed enhancements. DataFlow machines were originally conceived to realize high levels of instruction parallelism and often exploit alternative programming languages. This architecture forms the basis for the JavaFlow machine and modifications in both structure and objective are key to the success of the design. Finally the Java language offers a combination of complexity through its abstractions and also the opportunity for the JavaFlow machine to achieve optimizations due to its distributed processing capabilities.

Chapter 4: JavaFlow Overview

SECTION 4.1 - OVERVIEW

Before presenting a system diagram and associated descriptions, Figure 11 describes the overall process of translating a computing problem (application) to the circuitry which ultimately implements the solution. The focus/scope of this system includes the interface language, instruction set architecture, and machine structure. In this system, this intermediate language is the Java Virtual Machine (JVM). Similarly, the logic design, network implementation, and cache structures which are critical to the performance of any computing system are not the focus of this research.

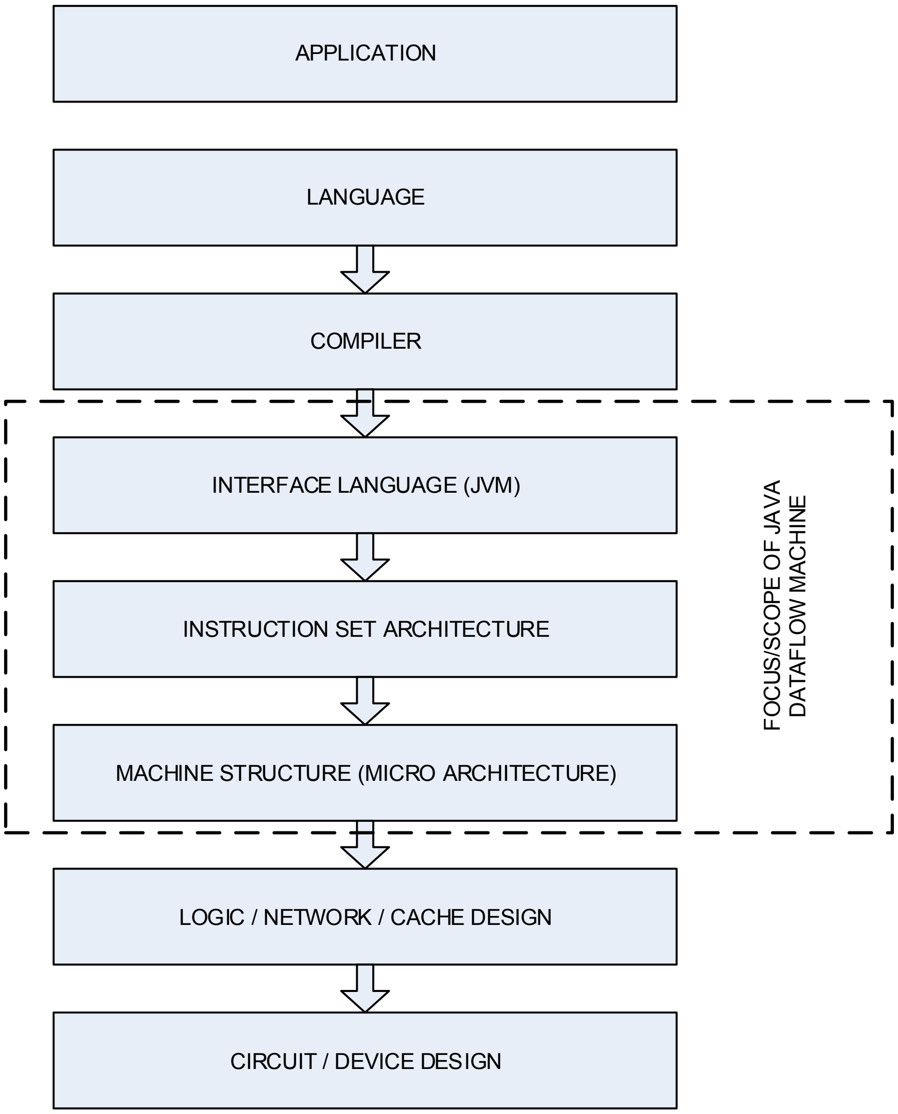

APPLICATION
LANGUAGE
COMPILER
INTERFACE LANGUAGE (JVM)
INSTRUCTION SET ARCHITECTURE
MACHINE STRUCTURE (MICRO ARCHITECTURE)
LOGIC / NETWORK / CACHE DESIGN
CIRCUIT / DEVICE DESIGN
FOCUS/SCOPE OF JAVA DATAFLOW MACHINE

Figure 11 Machine Architecture Layers

Chapter 3 provided a basic description of several DataFlow machines in addition to specific similar machine structures. In addition some key characteristics of the Java Virtual Machine have been described. Chapter 5 provides an analysis of Java Benchmarks demonstrating the characteristics of two sets of benchmarks and how a Java DataFlow machine might provide an effective execution platform. This chapter provides an overview description of this machine while Chapter 6 provides additional details. Note that since this machine description is still at a high level, there are some technology related decisions and details that are not fully defined. Examples of these are specific bus widths, tag widths, and machine state storage sizes. The advantage of combining the General Purpose Processor with the DataFlow Fabric is that when DataFlow technology limitations are met, processing can still proceed using the GPP.

Figure 12 shows the overall machine configuration with the 3 types of networks, the DataFlow fabric, the GPP, and Memory. To address the challenges of global wiring not scaling as technology [8], a Globally Asynchronous / Locally Synchronous (GALS) [48, 49] design is utilized. Each Instruction Node can have its own synchronous clock to control processing functions; however data transfers between Instruction Nodes can be asynchronous and can proceed a different rates on each network and on sections of each network. Furthermore, the transfer protocols of the serial networks can be optimized for the lack of requirement for routing to reduce data transfer delays.

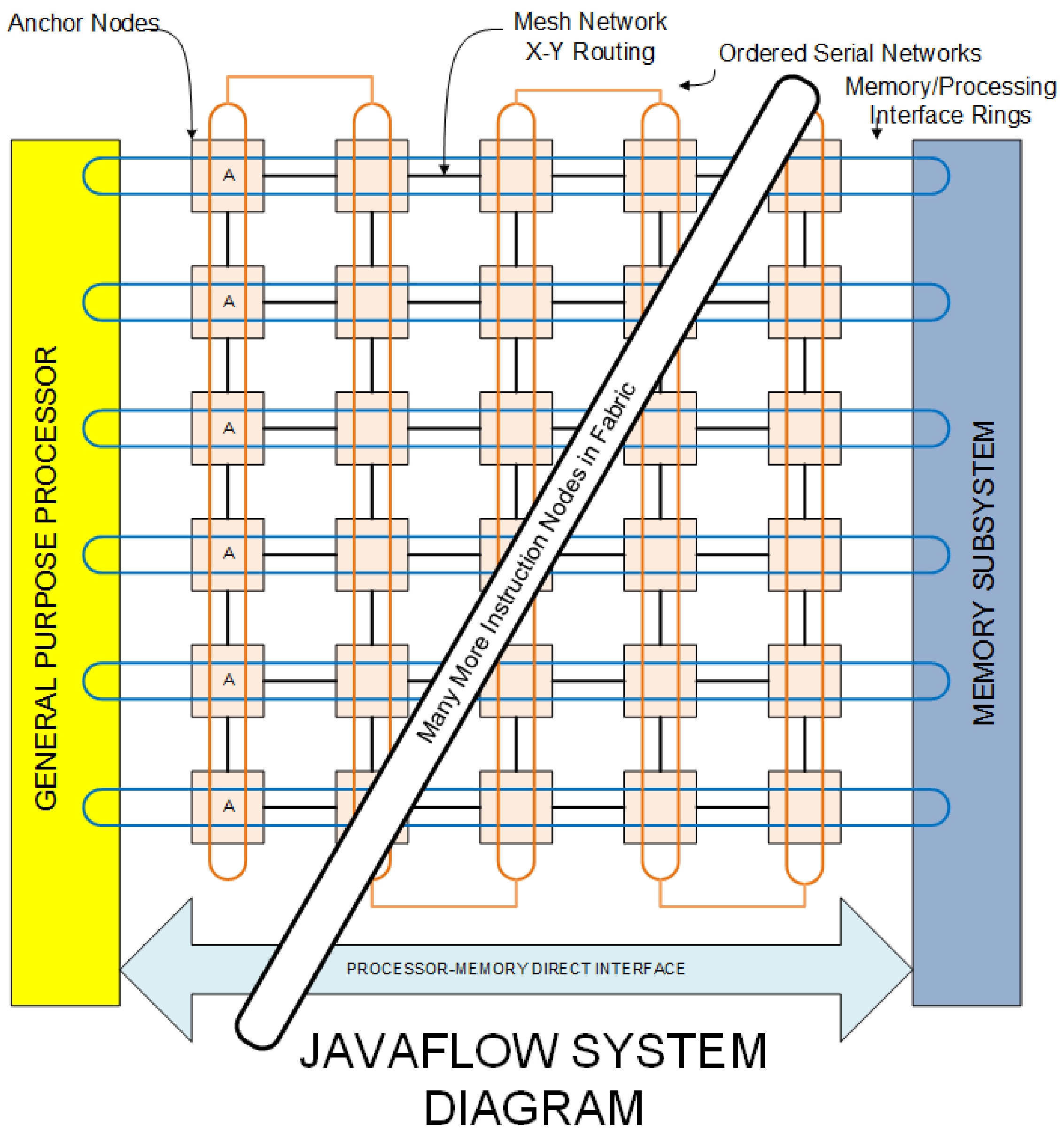
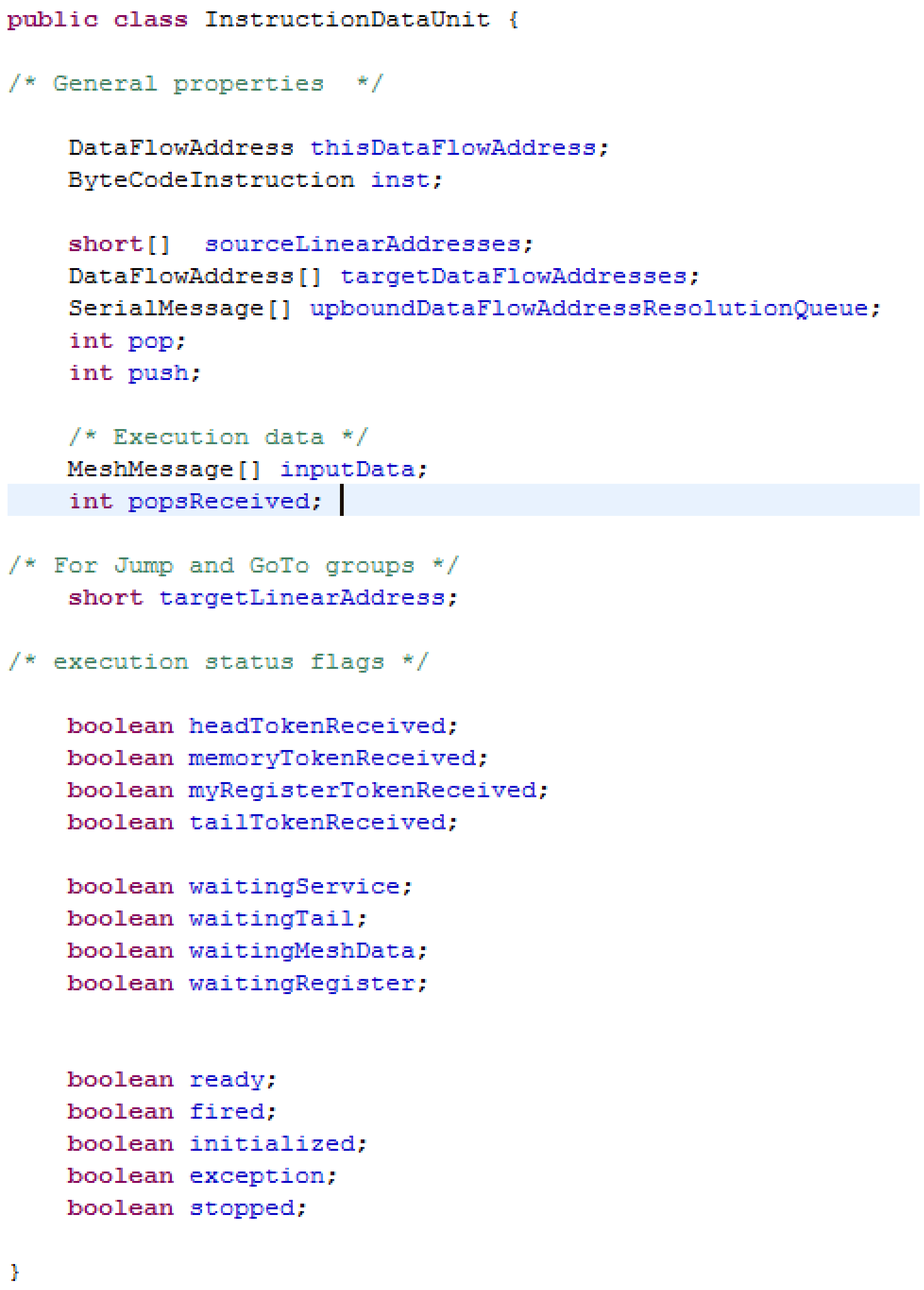


Figure 12 JavaFlow System Diagram

Section 4.2 - Instruction Nodes

The basic element in the DataFlow Fabric is the Instruction Node. Each Instruction Node has a unique (x, y) address in the Fabric and contains the following components:

- Instruction Execution Unit. This is the actual processor that executes the decoded instructions in the node. This unit interfaces to both the routers and to the Instruction Data Unit(s).
- Instruction Data Unit(s). Each Instruction Node has one or more Instruction Data Units which house the actual ByteCode instructions and all the associated state information for the ByteCode instruction. Figure 13 shows the resources for each Instruction Execution Unit which are explained as the overall instruction processing is described.
- Serial Network Router. The interface to the forward and reverse serial networks which connect each Instruction Node and is used to manage the DataFlow Fabric and insure proper control flow is maintained with the DataFlow machine structure. Serial Tokens are used to communicate between Instruction Nodes on behalf of each Instruction Data Unit.
- Mesh Network Router. The interface to each of the four adjacent Instruction Nodes and the function that sends mesh messages throughout the DataFlow Fabric.
- GPP/Memory Interface. For selected Instruction Nodes this function is the interface between the Instruction Execution Node and the high speed ring networks to carry memory data and request/responses to the GPP



```
public class InstructionDataUnit {

/* General properties  */

    DataFlowAddress thisDataFlowAddress;
    ByteCodeInstruction inst;

    short[]  sourceLinearAddresses;
    DataFlowAddress[] targetDataFlowAddresses;
    SerialMessage[] upboundDataFlowAddressResolutionQueue;
    int pop;
    int push;

    /* Execution data */
    MeshMessage[] inputData;
    int popsReceived;

/* For Jump and GoTo groups */
    short targetLinearAddress;

/* execution status flags */

    boolean headTokenReceived;
    boolean memoryTokenReceived;
    boolean myRegisterTokenReceived;
    boolean tailTokenReceived;

    boolean waitingService;
    boolean waitingTail;
    boolean waitingMeshData;
    boolean waitingRegister;


    boolean ready;
    boolean fired;
    boolean initialized;
    boolean exception;
    boolean stopped;

}
```

Figure 13 Instruction Data Unit Resources

Each Instruction Data Node knows its own physical (x, y, p) Mesh address where 'p' represents the number of the Instruction Data Unit internal to the Instruction Node. When instructions are loaded they know their own serial address (offset from the first instruction of the method) and the addresses of the next instructions to be executed. If the next instruction is ***sequential***, then the next instruction is the current instruction number incremented by one. If the instruction has a non-sequential next address, then the next serial address is included as part of the instruction. In the architecture for the ByteCode instructions, each instruction may have variable lengths, and instruction addressing is based on byte addresses in the loaded instruction stream. At this level of the JavaFlow design, all instructions are a single length and the linear addresses are independent of the size of the ByteCode instructions. Each Instruction Data Unit has its own unique serial address which is the absolute number of the ByteCode instruction.

The sourceLinearAddresses contained in the Instruction Data Unit are the addresses of instructions that transfer control to that Instruction Data Unit. These sourceLinearAddresses are used in the address resolution where the operand addressing is translated to the DataFlow Fabric addresses. The resulting set of target DataFlow addresses are stored in an array in the Instruction Data Unit. This address resolution process is described in Section 6.2.

The various status items are set when specific Serial and Mesh messages arrive and when the instruction actually executes or 'fires.' The 'pop' value is the number of stack elements that are consumed by the instruction. The push value is the number of stack (DataFlow) elements that are produced by the instruction. The'PopsReceived' value is a count of the number of data elements received so that when 'pop'=='PopsReceived', the instruction can fire.

The simplest case would be for each Instruction Node to house only a single ByteCode instruction. This would minimize any run time instruction decode and would allocate all resources in the Instruction Node to a single Instruction Execution Unit. However like previous distributed DataFlow machines each node is expected to house n instructions. A simple and reasonable value for this value is 64, although technology decisions may allow larger or smaller numbers. Note that these instructions could be from different methods or even different threads, as instructions are tagged with thread-method identifiers. The larger the number of instructions housed in each Instruction Node, the more complicated the control at each element becomes. If the number of instructions housed in each element were reduced to 1, then there would be more opportunity for single thread parallelism but with potentially longer mesh network transit times between Instruction Nodes. Allowing the number to grow to a very large number of instructions creates an implementation more similar to a modern multi-core machine where each core executes an entire instruction stream.

Instruction Nodes in the DataFlow Fabric can be heterogeneous. For example, for each 10 Instruction Nodes, 6 could be general purpose logic/arithmetic, 1 floating point, 2 storage, 1 control. Using a heterogeneous set would increase the transit times for data over the mesh network as would be distributed over a larger area of the chip. Results of methods loaded in this specific configuration are included in the performance analysis section of the Results: Chapter 7.

The goal of the JavaFlow machine is to implement the maximum number of Instruction Nodes consistent with technology ground rules. In addition, housing more instructions may be achieved with reasonable numbers of Instruction Data Units inside each Instruction node. For simplicity and to stress the DataFlow Fabric, the simulations in Chapter 7 utilize a single Instruction Data Unit in each Instruction Node.

SECTION 4.3 - LIMITATIONS

One of the key advantages of the JavaFlow machine is that the state data for each method is distributed throughout the DataFlow Fabric. Since this state data is finite and small, there are several side effects that may limit some of the methods that can be deployed. Like the TRIPS machine, each method must execute atomically. This means that the Anchor node, which is the first node of a method may not allow any subsequent execution of the method until the current thread exits. Furthermore, recursive calls are not allowed. A thread may be executing multiple methods at any given time, but each individual method may have only one thread active at a time.

The ByteCode instruction set has a series of special instructions that cannot be directly executed by the elements in the DataFlow Fabric. These special instructions must send messages to the GPP for assistance. The instruction mix data indicates that this should occur infrequently and have minimal impact on system performance. Specific instructions are described in Section 6.3.

Chapter 5: Benchmarks

SECTION 5.1 - OVERVIEW

Prior to the detailed definition of the JavaFlow machine, it was necessary to understand the basic structure of Java Methods in real applications and to determine if a DataFlow structure could be defined and optimized for their execution. The benchmark analysis had several major components:

- Size of method
- Each method's effect on the overall benchmark performance
- Dynamic instruction mix
- Static instruction mix
- Number of jumps (control flow events)
- Maximum register and stack requirements for each method

The size of each method will determine if the entire method could be resident in the DataFlow fabric. With each method being invoked a different number of times during the total benchmark, some selected methods were expected to account for the majority of the execution time of the overall benchmark. The following results demonstrated a surprisingly small number of methods had major leverage in the overall benchmark performance. The dynamic instruction mix will have a major role in the overall method's performance, while the static mix will determine which computing elements are needed to load the method.

Heterogeneous nodes in the DataFlow fabric will allow the best optimization of silicon real estate. For example a floating point arithmetic node would be significantly larger than an integer arithmetic or logical node. Therefore the static instruction mix allows the establishment of the proper number of each nodes in the fabric.

As described in the following section, branches and especially backwards branches in the control flow require special handling in the DataFlow Fabric. The number and location of these branches have higher impact on the method's performance than in a typical von Neuman processor.

Finally, a characteristic of the Java language is that the maximum number of registers and stack elements are known before the ClassFile is loaded. Since one of the key aspects of the JavaFlow machine is to maintain machine state throughout the DataFlow Fabric, this number is key to defining some of the intermediate storage requirements that certain nodes must possess.

While an understanding of these parameters are key to the overall effectiveness of the JavaFlow machine, unusual values for some methods means that those methods must be executed in a more traditional manner utilizing the General Purpose Processor.

The Standard Performance Evaluation Corporation (SPEC) [50] publishes a series of performance benchmarks for use in a wide variety of computing applications. For the JVM implementation of the JavaFlow machine, the SpecJVM98 and SpecJVM2008 benchmarks were utilized. The older SpecJVM98 was used to compare against existing literature while the modern SpecJVM2008 was also analyzed. These benchmarks provide a series of Java Class files to measure the performance of the JVM. In addition each release provides a 'harness' which executes the series of benchmarks and reports results. Appendix D provides lists of all the included and excluded benchmarks from each set along with brief descriptions of each benchmark.

SECTION 5.2 - DYNAMIC MIX METHODOLOGY AND RESULTS

To both develop design parameters for the JavaFlow machine and to evaluate its performance, an existing JVM was instrumented to provide analysis of both SPEC JVM benchmarks. JAMVM [51] is a relatively small JVM offering both an interpreter and JIT variant. Release 1.5.3 was modified to collect method usage and ByteCode instruction mix information. JAMVM uses the GNU Classpath code which results in the reported benchmarks being a subset of the total SpecJVM2008 set [52].

The methodology of this analysis was to establish a 256 element array for each method signature which was executed. Each element in the array is a counter for the corresponding ByteCode instruction. These arrays are generated while the program is run and then processed after the benchmark completes to generate the data presented here. For the primary analysis, the JAMVM machine was used in its interpreter mode and the benchmark configured to run a single thread. The SPEC2008 runs were run only for 2 iterations and therefore are not fully compliant, but since only mix data was being generated, cache warmup characteristics were not critical. Also the dominant processing was in the actual benchmark code even though the 'harness' methods were also included in the analysis. This may skew the number of methods invoked, but the mix analysis remains valid.

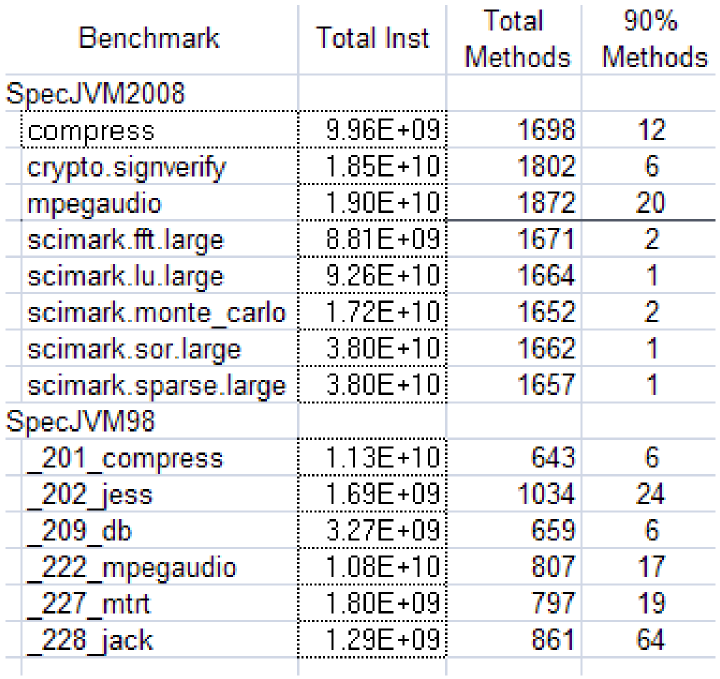
Since this analysis was focused on capturing significant amounts of instruction executions over several iterations of each benchmark, trace data was not gathered. This results in assumptions being made in the Results: Chapter 7, regarding branch behavior.

Based on work by Radhakrishnan [53] and confirmed with this analysis, the majority of the total number of ByteCode instructions executed are found in a small number of methods invoked in the benchmark. Table 1 shows selected SPEC benchmarks with the total instructions executed; the total number of methods utilized; and finally the number of methods that encompass 90% of the number of instructions executed. There is no hard scientific rationale for choosing the 90% number for the data in the last column in Table 1. The number of methods contributing to the 90% of the overall performance appear to be small enough to analyze more completely, and results in future Sections shows that these will fit inside a DataFlow Fabric.

Note that the more modern SpecJVM2008 benchmarks have fewer methods at the 90% level than SpecJVM98, and that all 5 of the scientific benchmarks have only 1 or 2 methods which determine the overall performance.

The implication of this result is that deploying a few types of methods in an application to a special purpose hardware subsystem like the DataFlow Fabric can have a significant effect on the overall performance. This is the same strategy which is used by JIT compilers which focus on compiler optimizations on only those methods with highly repetitive execution.

Table 1 Method Utilization in SPEC Benchmarks



Benchmark	Total Inst	Total Methods	90% Methods
SpecJVM2008			
compress	9.96E+09	1698	12
crypto.signverify	1.85E+10	1802	6
mpegaudio	1.90E+10	1872	20
scimark.fft.large	8.81E+09	1671	2
scimark.lu.large	9.26E+10	1664	1
scimark.monte_carlo	1.72E+10	1652	2
scimark.sor.large	3.80E+10	1662	1
scimark.sparse.large	3.80E+10	1657	1
SpecJVM98			
_201_compress	1.13E+10	643	6
_202_jess	1.69E+09	1034	24
_209_db	3.27E+09	659	6
_222_mpegaudio	1.08E+10	807	17
_227_mtrt	1.80E+09	797	19
_228_jack	1.29E+09	861	64

Table 2 shows the dynamic instruction mix for those methods comprising the 90% of the execution cycles of the benchmarks. This is the dynamic mix that would be executed in the DataFlow fabric.

Table 2 Dynamic Instruction Mix of 90% Methods

Benchmark	Locals+Stack	Arith-Fixed	Arith-Float	Branch	Calls-Returns	Constants	Constants-Stg	Object+Special	Storage-Ordered
SpecJvm2008 (total)	47%	10%	11%	10%	1%	3%	0%	0%	19%
compress	37%	15%	0%	8%	8%	8%	0%	0%	24%
crypto.signverify	43%	32%	0%	3%	0%	9%	4%	0%	9%
mpegaudio	38%	11%	7%	7%	1%	12%	1%	0%	24%
scimark.fft.large	54%	13%	11%	3%	0%	8%	0%	0%	11%
scimark.lu.large	50%	6%	12%	13%	0%	0%	0%	0%	19%
scimark.monte_carlo	29%	8%	8%	11%	4%	5%	1%	0%	33%
scimark.sor.large	46%	9%	17%	6%	0%	6%	0%	0%	17%
scimark.sparse.large	54%	6%	11%	12%	0%	1%	0%	0%	17%
SpecJvm98 (total)	39%	9%	4%	7%	6%	9%	0%	0%	25%
_201_compress	44%	11%	0%	8%	4%	7%	0%	0%	26%
_202_jess	40%	3%	0%	13%	13%	6%	1%	3%	22%
_209_db	40%	12%	0%	10%	9%	0%	0%	2%	26%
_222_mpegaudio	36%	8%	11%	3%	1%	14%	1%	0%	26%
_227_mtrt	26%	0%	9%	4%	31%	7%	0%	0%	23%
_228_jack	37%	6%	0%	12%	15%	7%	0%	2%	21%

Analysis of the dynamic instruction mix is key to JavaFlow performance since one of the follow on enhancements to the JavaFlow machine is an expected performance advantage through reduction of ByteCode instructions. This process of folding has been demonstrated [18], and due to the nature of the data flow machine, instructions moving data to/from the stack and local storage could be eliminated directly. Also stack movement instructions could be eliminated. The analysis reported in Chapter 7 does not account for this folding enhancement.

The 'Locals+Stack' column which represents 26% to 54% of the total instructions are all candidates for being combined or 'folded' into other instructions. This is a suggested follow-on enhancement described in Section 6.4. The arithmetic instructions are split by fixed and floating point to understand allocation of arithmetic resources. The SpecJVM2008 benchmarks utilize fewer calls than the older SpecJVM98 benchmark.

The 'Constants-Stg' represents constants coming from the local variable pool and can be unordered accesses to memory where the larger number of array and field memory operations must be ordered. The 'Object+Special' instructions represent a small number of operations that require the support of the general purpose processor.

To further amplify the point about a small number of methods contributing to a major percentage of the performance, Table 3 and Table 4 show the contribution to the performance of the top 4 methods in both the SpecJvm2008 and SpecJvm98 benchmarks. The classes and methods are listed with the percentage for each method. The percentage to the right of the table is the sum of these 4 methods against the total number of operations executed in each benchmark. There is a wide range, however in 7 of the 14 benchmarks, the top 4 methods account for over 80% of the instructions, and in 3 benchmarks, a single method accounts for 99% of the performance.

Table 3 SpecJvm2008 - Top 4 Methods

	BM	Class-Method	Total Ops	%	Top 4
SpecJvm2008			2.82E+11		
	compress		9.96E+09		55%
		spec/benchmarks/compress/Compressor .compress	2.59E+09	26%	
		java/util/zip/CRC32 .update	1.16E+09	12%	
		spec/benchmarks/compress/Decompressor .decompress	9.20E+08	9%	
		spec/benchmarks/compress/Compressor .output	8.22E+08	8%	
	crypto.signverify		1.85E+10		83%
		gnu/java/math/MPN .submul_1	4.58E+09	25%	
		gnu/java/security/hash/Sha160 .sha	4.47E+09	24%	
		gnu/java/security/hash/Sha256 .sha	3.57E+09	19%	
		gnu/java/math/MPN .mul	2.71E+09	15%	
	mpegaudio		1.90E+10		31%
		javazoom/jl/decoder/LayerIIIDecoder .dequantize_sample	1.76E+09	9%	
		javazoom/jl/decoder/LayerIIIDecoder .inv_mdct	1.55E+09	8%	
		javazoom/jl/decoder/huffcodetab .huffman_decoder	1.32E+09	7%	
		javazoom/jl/decoder/LayerIIIDecoder .hybrid	1.31E+09	7%	
	scimark.fft.large		8.81E+09		96%
		spec/benchmarks/scimark/fft/FFT .transform_internal	7.69E+09	87%	
		spec/benchmarks/scimark/fft/FFT .bitreverse	4.78E+08	5%	
		spec/benchmarks/scimark/utils/Random .nextDouble	1.67E+08	2%	
		spec/benchmarks/scimark/fft/FFT .inverse	1.01E+08	1%	
	scimark.lu.large		9.26E+10		100%
		spec/benchmarks/scimark/lu/LU .factor	9.18E+10	99%	
		spec/benchmarks/scimark/utils/Random .nextDouble	1.67E+08	0%	
		spec/benchmarks/scimark/utils/kernel .matvec	1.26E+08	0%	
		spec/benchmarks/scimark/utils/kernel .CopyMatrix	8.61E+07	0%	
	scimark.monte_carlo		1.72E+10		99%
		spec/benchmarks/scimark/utils/Random .nextDouble	1.33E+10	77%	
		spec/benchmarks/scimark/monte_carlo/MonteCarlo .integrate	3.65E+09	21%	
		gnu/java/math/MPN .submul_1	2.99E+07	0%	
		gnu/java/security/hash/Sha160 .sha	2.02E+07	0%	
	scimark.sor.large		3.80E+10		100%
		spec/benchmarks/scimark/sor/SOR .execute	3.75E+10	99%	
		spec/benchmarks/scimark/utils/Random .nextDouble	1.67E+08	0%	
		spec/benchmarks/scimark/utils/kernel .RandomizeMatrix	6.29E+07	0%	
		gnu/java/math/MPN .submul_1	2.99E+07	0%	
	scimark.sparse.large		3.80E+10		99%
		spec/benchmarks/scimark/sparse/SparseCompRow .matmult	3.75E+10	99%	
		spec/benchmarks/scimark/utils/Random .nextDouble	1.67E+08	0%	
		spec/benchmarks/scimark/sparse/SparseCompRow .measureSparseMatmult	5.78E+07	0%	
		spec/benchmarks/scimark/utils/kernel .RandomVector	4.20E+07	0%	

Table 4 SpecJvm98 - Top 4 Methods

SpecJvm98-100cmd			3.02E+10		
	_201_compress		1.13E+10		76%
		spec/benchmarks/_201_compress/Compressor .compress	3.96E+09	35%	
		spec/benchmarks/_201_compress/Decompressor .decompress	2.65E+09	24%	
		spec/benchmarks/_201_compress/Compressor .output	1.01E+09	9%	
		spec/benchmarks/_201_compress/Input_Buffer .getbyte	9.41E+08	8%	
	_202_jess		1.69E+09		43%
		spec/benchmarks/_202_jess/jess/Node2 .runTests	2.15E+08	13%	
		spec/benchmarks/_202_jess/jess/ValueVector .equals	1.94E+08	12%	
		spec/benchmarks/_202_jess/jess/Value .equals	1.82E+08	11%	
		spec/benchmarks/_202_jess/jess/Token .data_equals	1.34E+08	8%	
	_209_db		3.27E+09		84%
		java/lang/String .compareTo	1.35E+09	41%	
		spec/benchmarks/_209_db/Database .shell_sort	8.89E+08	27%	
		java/util/Vector .elementAt	3.15E+08	10%	
		java/util/Vector .checkBoundExclusive	1.80E+08	6%	
	_222_mpegaudio		1.08E+10		62%
		spec/benchmarks/_222_mpegaudio/q .l	4.70E+09	43%	
		spec/benchmarks/_222_mpegaudio/q .m	8.07E+08	7%	
		spec/benchmarks/_222_mpegaudio/lb .read	6.30E+08	6%	
		spec/benchmarks/_222_mpegaudio/cb .Ä£	5.30E+08	5%	
	_227_mtrt		1.80E+09		48%
		spec/benchmarks/_205_raytrace/OctNode .Intersect	3.44E+08	19%	
		spec/benchmarks/_205_raytrace/Point .Combine	2.00E+08	11%	
		spec/benchmarks/_205_raytrace/OctNode .FindTreeNode	1.98E+08	11%	
		spec/benchmarks/_205_raytrace/Face .GetVert	1.22E+08	7%	
	_228_jack		1.29E+09		17%
		spec/benchmarks/_228_jack/RunTimeNfaState .Move	7.41E+07	6%	
		spec/benchmarks/_228_jack/TokenEngine .getNextTokenFromStream	5.11E+07	4%	
		java/lang/String .<init>	5.07E+07	4%	
		java/util/Hashtable$EntryEnumerator .nextElement	4.71E+07	4%	

A final piece of information derived from the dynamic mix analysis involved the utilization of '_Quick' instructions. These instructions referenced in Section 3.6 are not part of the official JVM definition but are utilized by interpreters to optimize the performance of storage instructions. Specifically these instructions include the actual pointer to data after the process of resolving this pointer by accessing the Constant Pool and any other Heap storage management functions have been performed. The JavaFlow machine simulation is based on the use of these resolved addresses, so it is key to understand the ratio of their use in the dynamic mix versus the original instructions that would require lookup actions.

Table 5 shows that 97% and 99% of the dynamic storage access instructions have been resolved and executed as '_Quick' instructions. This provides assurance that the assumptions in the Results: Chapter 7 are valid by excluding the time to resolve the addresses from the overall execution time. Note that in the machine description, the assertion is made that the distributed processing capability of the JavaFlow machine can resolve these addresses more effectively than traditional means is key to the continued use of cached methods with different objects that would require additional translations.

Table 5 Impact of Quick Instructions

Benchmark	Total Ops	Storage Base	Storage Quick	Percentage
SpecJvm2008	$2.82\text{x}10^{11}$	$3.38\text{x}10^{8}$	$1.08\text{x}10^{10}$	97%
SpecJvm98	$3.02\text{x}10^{10}$	$3.83\text{x}10^{7}$	$5.90\text{x}10^{9}$	99%

SECTION 5.3 - STATIC MIX METHODOLOGY AND RESULTS

The generation of static mix data along with other control and dataflow analysis was conducted on the class files of the benchmarks. Since the class file format is complex, three tools were utilized to extract information from the archive files (JAR files) from the benchmarks:

1. BCEL [54] (Byte Code Engineering Library)
2. ASM [55] from the ObjectWeb Consortium
3. JAVAP – Oracle Java ClassFile disassembler which is part of the Oracle Java Development System.

The first two tools are Java Programs which accept Java ClassFiles as input and provide as series of analysis capabilities. Each also provides the opportunity to modify the ClassFile which can be the final stage of an optimization process. The newer, ASM system uses either an inheritance pattern or a delegation pattern to analyze and manipulate the Java ClassFiles. The JAVAP program simply outputs the disassembled ByteCode instruction stream and analysis was done with external software written for this project.

The static analysis performed on the benchmarks has primarily used both the ASM and the JAVAP software. In each case the class files were translated into readable assembly language files that could be processed. The ASM system outputs to the Jasmine [56] Java assembler format while JAVAP outputs to human readable format only. The Jasmine language which is described in Meyer's book is a way to learn and manipulate Java ByteCode statements without the complexity of the class file format. In both cases, analysis software was written to consume the output text files.

The goals of this part of the static benchmark analysis were to analyze each of the Java Methods in the benchmark, while focusing on those methods which comprise 90% of the performance impact as shown by the previous work. The desired results included static instruction mix, maximum stack and local variable usage, and control flow information associated with the number of basic blocks in the method and the number of back branches. Back branches are the result of loops and must be handled specially in the JavaFlow machine, so the number of these are key to the overall system performance. Finally, some data flow analysis was performed to understand the amount of stack information which is 'live' (valid) between basic blocks of code. Again, the JavaFlow machine would require some special handling for situations like this.

Table 6 Static Mix Analysis

Combined Static Mix Numbers:

Benchmark	%Arith	%Float	%Control	%Storage	Total Ops
SpecJvm2008	69%	14%	5%	19%	12,652
compress	63%	11%	14%	23%	766
crypto.signverify	91%	37%	1%	8%	2,998
mpegaudio	62%	7%	5%	24%	8,082
scimark.fft.large	73%	11%	7%	11%	337
scimark.lu.large	69%	7%	12%	15%	162
scimark.monte_carlo	51%	5%	14%	23%	110
scimark.sor.large	70%	9%	8%	14%	111
scimark.sparse.large	64%	7%	17%	15%	86
SpecJvm98-100cmd	55%	3%	15%	23%	10,083
_201_compress	65%	11%	11%	24%	783
_202_jess	53%	2%	28%	18%	844
_209_db	63%	8%	16%	22%	209
_222_mpegaudio	55%	2%	3%	28%	4,169
_227_mtrt	50%	0%	28%	16%	1,917
_228_jack	55%	5%	23%	21%	2,161
Total	63%	9%	9%	21%	22,735
Conclusion:	60%	10%	10%	20%	

The instructions have been grouped into 4 primary types which may be the types of heterogeneous elements in the DataFlow Fabric. The detailed results are shown for the methods that comprise 90% of the performance as described in the previous analysis for each benchmark and then summarized for the two benchmark groups. The conclusion line shows an approximate average of all the data indicating the mix of elements that should be implemented in the DataFlow Fabric, assuming that all homogeneous elements was impractical.

The final column of Table 6 shows the total number of instructions in each of these benchmarks. Notice that of the 14 benchmarks, 9 have instruction counts less than 1000, while 5 have greater than 1000 instructions. Depending on the expected level of multi-threading, it appears reasonable that each of the 9 benchmarks could be resident, and perhaps all simultaneously. If the entire 23K instructions were to be resident in the DataFlow Fabric, then the approximate cube root of the total would yield a 30 by 30 element fabric with each Instruction Node holding 30 instructions. This would support 27,000 resident instructions.

Finally, although the 90% of the performance is driven by 23,000 instructions, the total number of static instructions in all benchmarks number 128,929. That means that 20% of the static instructions account for 90% of the dynamic performance.

SECTION 5.4 - DATAFLOW AND CONTROLFLOW ANALYSIS

The final stage of the benchmark analysis focused on the DataFlow and ControlFlow of the top 10% of the Spec benchmarks. The objective is to translate the ByteCode methods into a DataFlow graph and understand the way data is transferred among the instructions and to identify any serious problems that may preclude an optimal design of the JavaFlow machine. This analysis was conducted by implementing a simulation of the JavaFlow class loader so that the ByteCode methods were actually installed into a DataFlow fabric and then messages passed serially up the control flow of the method to resolve addresses. This is the translation required to implement the DataFlow representation of the program when starting with the procedural method. Note that the specific implementation of this process is described in detail in Section 6.1 where the JavaFlow machine description is provided.

The JavaFlow machine handles register transfers via a serial bus to order their read and write actions based on the control flow of the machine. A challenge is the stack variables that may be passed to multiple forward destinations (e.g., a dataflow merge) or one or more back destination (e.g., dataflow back). Dataflow merges can be handled by implementing multiple destination addresses in each instruction, but back merges present a serious problem.

A DataFlow merge is where a DataFlow instruction side has 2 source instructions. A DataFlow back merge is a DataFlow merge where one of the source instructions is further down the control flow of the instruction stream. The only way this could possibly occur is in with a jump back. However, with the restrictions on the stack described in Section 3.6, it is unlikely that a valid Java program could create such a condition. However, the analysis was performed to insure this condition.

Table 7 shows the results of the JavaFlow class loader simulator in the area of branches, merges and back branches. Note that in the benchmarks, there are NO back merges which means that only registers are used to bring data back through the control flow. This might be expected from both compiler design experience and the restrictions on the Java Virtual Machine. Compiler optimizations typically utilize registers for transferring data between basic blocks while using stack variables for transfers within blocks. Furthermore the JVM restricts all inputs to a block to have the same stack signature which further enforces the above compiler design.

The ‘Total Instructions’ column is similar to Table 6, and the ‘Forward’ and ‘Back’ columns count the number of control flow branches in each method. (Note: As in previous tables, this is a summary of all the methods within each benchmark which comprise the top 10% of the dynamic performance). The ‘Total DFlows’ column represents the number of times translations are required from the procedural steps to the DataFlow instructions. The ‘Total DFlows Merge’ are examples of where stack variables need to be sent to multiple downstream branches. The key element in this analysis is that there are no instances of back merges ‘Total DFlows Back.’

Finally the 'Total Cycles' column is the result of an initial simulation of the address resolution process and shows that by using the serial busses described in the next section, that this critical process can be completed in approximately twice the number of byte code instructions loaded.

Table 7 Benchmark DataFlow and Control Flow Analysis

Benchmark	Forward	Back	Total Insts	Total Cycles	Total DFlows	Total DFlows Merge	Total DFlows Back
SpecJvm2008	320	116	12652	25552	10659	17	0
compress	47	10	766	1591	572	2	0
crypto.signverify	19	15	2998	6034	2628	0	0
mpegaudio	218	71	8082	16275	6847	15	0
scimark.fft.large	9	6	337	686	270	0	0
scimark.lu.large	10	5	162	331	123	0	0
scimark.monte_carlo	8	1	110	228	83	0	0
scimark.sor.large	4	4	111	230	81	0	0
scimark.sparse.large	5	4	86	177	55	0	0
SpecJvm98-100cmd	492	71	9885	20255	7423	32	0
_201_compress	43	7	783	1602	600	0	0
_202_jess	64	13	645	1358	384	2	0
_209_db	15	5	209	435	141	2	0
_222_mpegaudio	52	13	4169	8429	3682	10	0
_227_mtrt	119	8	1917	3905	1215	0	0
_228_jack	199	25	2162	4526	1401	18	0
Sum	812	187	22537	45807	18082	49	0

SECTION 5.5 - SUMMARY

Table 8 summarizes the analysis which shows the performance of complex benchmarks is dependent on a small number of methods, and further investigation showed that the number of instructions, registers, and branches would fit within the DataFlow fabric.

Table 8 Analysis Summary

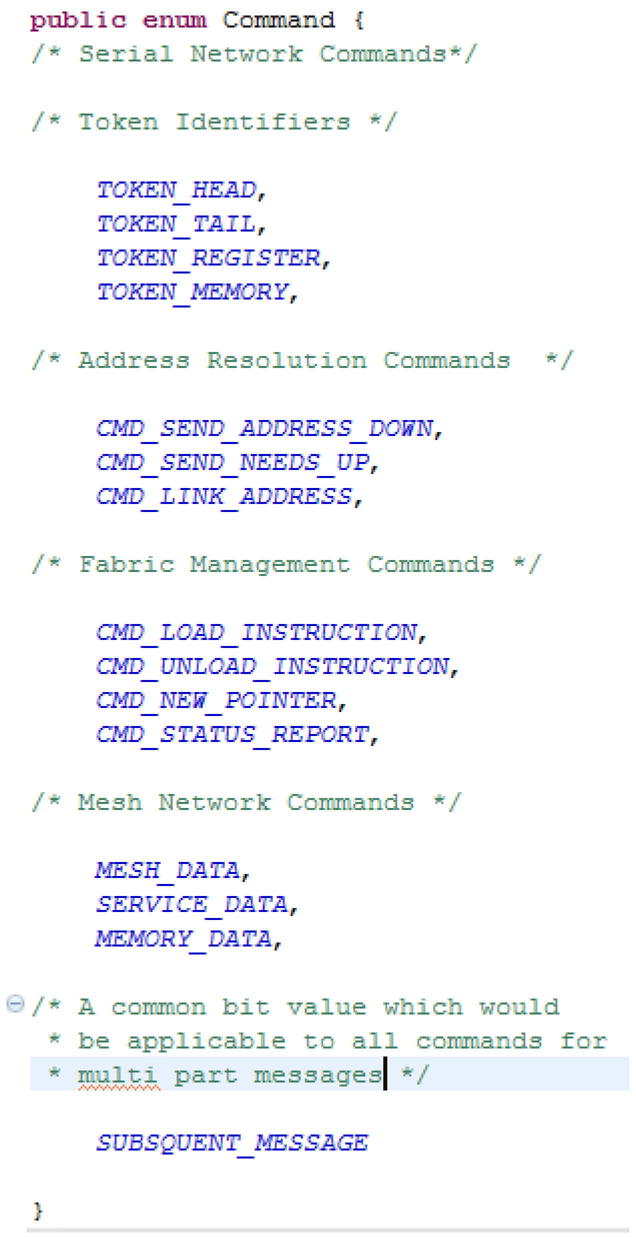
Dynamic Methods Executed	18,479
Dynamic Instructions Executed	$2.7*10^{11}$
Methods taking 90% total Time	181
Methods Analyzed	160
Avg. Inst/Method	71
Avg. Registers/Method	6
Static Mix	
Arithmetic	60%
Floating Pt	10%
Control	10%
Storage	20%
Average # Forward Branches	4.6
Average # Back Branches	1

The analysis preceding the design of JavaFlow is based on the use of a set of industry standard benchmarks to understand both the dynamic execution profile of Java methods and their ByteCode instructions along with static analysis of the same methods. A small number of methods comprise a significant percentage of the instructions executed in each benchmark, and some of the more obscure Java ByteCode instructions are not present in the code. DataFlow and ControlFlow analysis demonstrates that entire methods can be housed in the DataFlow fabric with need for assist from the General Purpose Processor to handle special control flow issues.

Chapter 6: JavaFlow Detailed Description

SECTION 6.1 - FUNCTIONAL UNITS

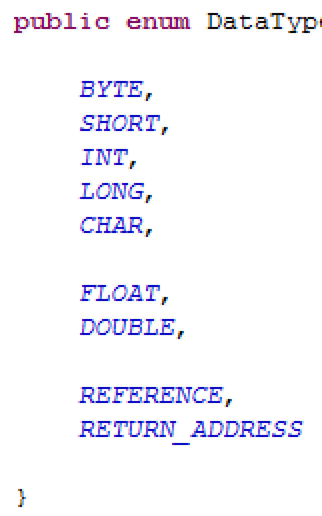
This section of the Machine Description goes into more detail of each of the functional units of the JavaFlow machine. In all subsequent descriptions of the resources of the JavaFlow machine, Java Class and Enum structures are used to describe the components. For example, both the Serial and Mesh Network have a field in their messages for commands. A summary of the command values is described by a Java Enum as in Figure 14. The use of these commands is described in the following sections.



```
public enum Command {
/* Serial Network Commands*/

/* Token Identifiers */

    TOKEN_HEAD,
    TOKEN_TAIL,
    TOKEN_REGISTER,
    TOKEN_MEMORY,

/* Address Resolution Commands  */

    CMD_SEND_ADDRESS_DOWN,
    CMD_SEND_NEEDS_UP,
    CMD_LINK_ADDRESS,

/* Fabric Management Commands */

    CMD_LOAD_INSTRUCTION,
    CMD_UNLOAD_INSTRUCTION,
    CMD_NEW_POINTER,
    CMD_STATUS_REPORT,

/* Mesh Network Commands */

    MESH_DATA,
    SERVICE_DATA,
    MEMORY_DATA,

/* A common bit value which would
 * be applicable to all commands for
 * multi part messages */

    SUBSQUENT_MESSAGE

}
```

Figure 14 JavaFlow Network Commands

Similarly both Serial and Mesh messages also contain a field describing the type of data in the payload which is shown in Figure 15. This is key to the strongly typed aspect of Java and insures that no dangerous pointer manipulation is allowed. There may be a need to expand this value to add qualifiers to the reference type to protect against aliasing which is where a class is reassigned to another class, and there may be both memory conflicts and possible type violations.



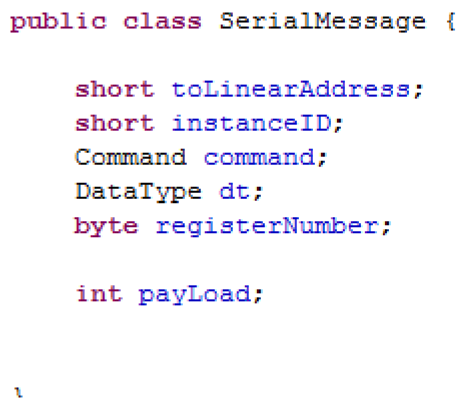
```
public enum DataType {

    BYTE,
    SHORT,
    INT,
    LONG,
    CHAR,

    FLOAT,
    DOUBLE,

    REFERENCE,
    RETURN_ADDRESS

}
```

Figure 15 JavaFlow DataTypes

Serial Network

The Serial Network is the key element of the machine which preserves the necessary control flow ordering of processing functions while allowing other operations to proceed with only data flow dependencies. It is also critical to the management of the DataFlow Fabric by loading instructions and resolving addresses. The key data transfer element in the Serial Network is a Token which is a specific type of Serial Message.

Serial Messages are described in Figure 16. As was described in the section on Java, each ByteCode instruction has only a linear address and in the case of jump and GoTo instructions, the linear address of the taken path. All other paths implicitly proceed to the next sequential instruction in the linear address space. The only routing function on the Serial Network is to send a message to the next Instruction Node in the linear sequence.



```
public class SerialMessage {

    short toLinearAddress;
    short instanceID;
    Command command;
    DataType dt;
    byte registerNumber;

    int payLoad;


}
```

Figure 16 Serial Message Structure

Therefore the Serial Messages do not have to carry the larger x, y, p coordinates of the DataFlow Mesh Fabric. A special ‘toLinearAddress’ of ‘Next’ would be used for most communications. Control flow changes would then use the actual target address contained in the instruction for the ‘toLinearAddress.’

In all cases the instanceID tag must be included in messages so that only those instructions involved in the current Thread, Class, Method, and Instance receive the command and data. Since this network is used to propagate all local registers to Instruction Nodes, the target register must be identified. For efficiency, this value may be combined with the command field. Finally a type field is include for run time validation. This insures that data of the proper type and width is being directed to specific Instruction Nodes, and allows exceptions to be generated in the case of mismatches.

Finally the payload of the message is envisioned to be a 32 bit element. Some technology implementations might support a 64 bit payload in a single transfer, but more likely any 64 bit entries would be broken in to multiple messages with the special command of SUBSEQUENT_MESSAGE used. Since both the Serial and DataFlow Mesh Networks used fixed routing schemes, there is no chance that the ordering of two consecutive messages from Node ‘a’ to Node ‘b’ would be violated.

There are a series of Tokens used for key tasks performed in the DataFlow Fabric. Tokens are defined primarily by the command field. The major groups of Tokens along with specific examples are:

- Instruction Load and Address Resolution
    - INSTRUCTION_TOKEN. The data element(s) that actually contains instructions that are loaded in to the Instruction Data Unit of Instruction Nodes.
    - ADDRESS_RESOLUTION_TOKEN. The set of commands to resolve addresses listed in Figure 14.
- Instruction Execution. These are described in the subsequent sections and are used to execute each instruction.
    - HEAD_TOKEN
    - MEMORY_TOKEN
    - REGISTER_TOKEN
    - TAIL_TOKEN
- Special Conditions and Management. These were not simulated, and are part of the overall management of the DataFlow Fabric and in handling special conditions
    - EXCEPTION_TOKEN
    - QUIESE_TOKEN. (To stop execution to allow Garbage collection or other special event.)

- RESETADDRESS_TOKEN. Part of the process of re-accessing the Constant Pool to get new pointers after a Garbage Collection event.

Figure 17 shows the way the Forward and Reverse Ordered Networks interface to the Instruction Node. The key aspect of this configuration is that all serial messages can interact with all Instruction Nodes for the specific Thread Class-Method-Instance being executed at a given time. Detailed message sequences are described in both the section on DataFlow Fabric Management where instructions are loaded and unloaded from the system, and again in the section describing the instruction execution.

Note that the Serial Network connects all nodes in the DataFlow Fabric. The top instruction in each method would be loaded immediately after the anchor node. Each instruction when loaded would have special flag indicating whether it was the bottom instruction of the method so that the Serial Network for the method would appear as a loop. As the size of the DataFlow Fabric increases, creative topologies can be explored on how to most effectively route the Serial Network to maximize both the utilization of all nodes and achieve maximum performance.

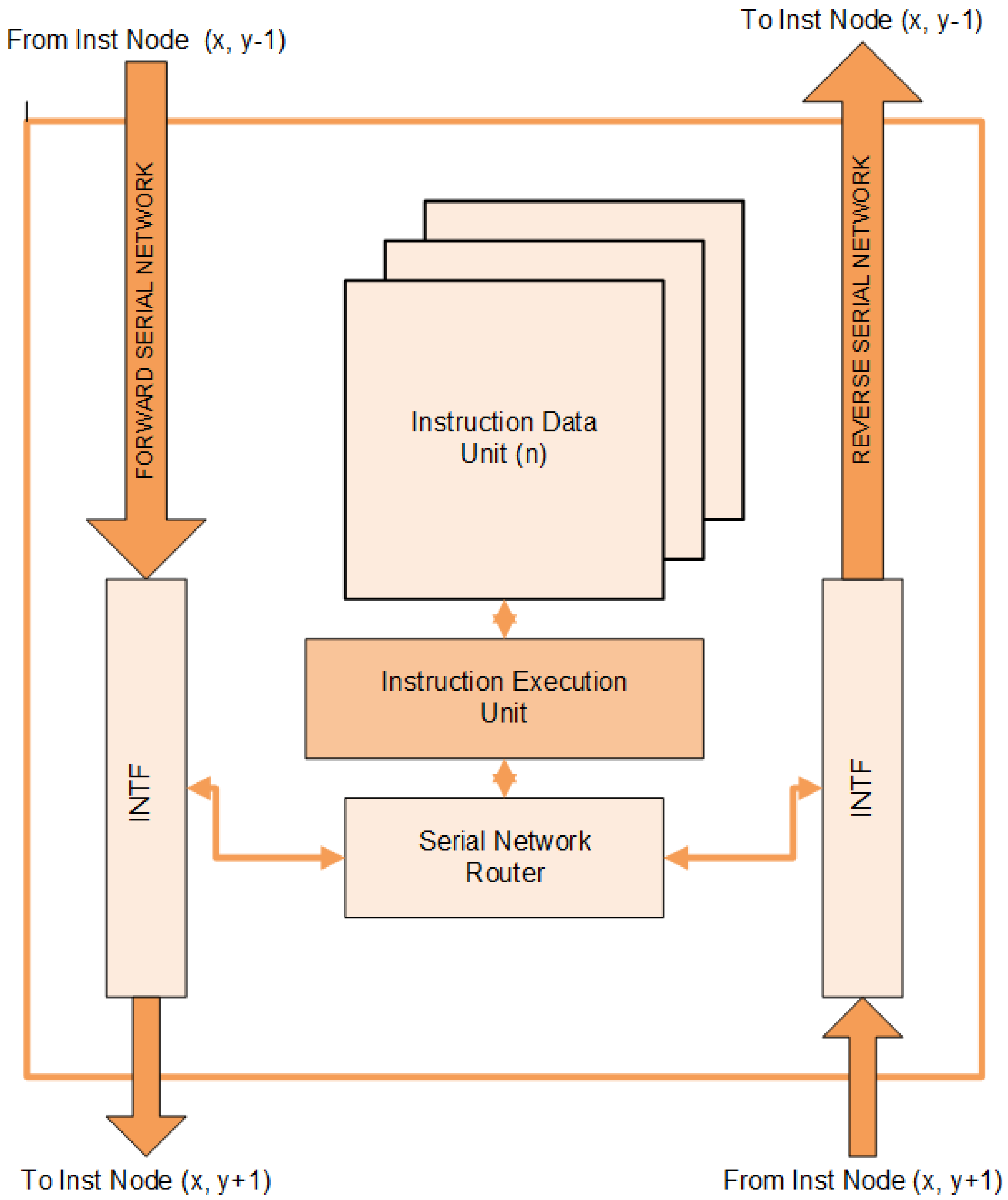


Figure 17 Serial Network Interface

DataFlow Fabric (Mesh Network)

The mesh network shown in Figure 18 is used for the traditional DataFlow Producer-Consumer data transfers. X-Y routing is used to insure no deadlocks. Unlike many on-chip networks, the length of data transfer is usually small. Initial simulations showed that a simple mesh performed better than a toroid if the addresses were assigned programmatically. The automatic routing in the JavaFlow should still work best with a simple network, although follow-on simulations with more complex structures may eliminate data edge transfer delays.

Since the routing function in the Mesh Network is more complex, it is envisioned that the data would move more slowly than in the Serial Network. Sensitivity to these clock relationships are part of the performance analysis in Chapter 7. Since the Serial Network is used to maintain appropriate control flow ordering and loops involve a control flow stall, the number of messages destined for a specific mesh node is finite and small. This means that a message level ACK protocol is not necessary which enhances performance and relieves network congestion.

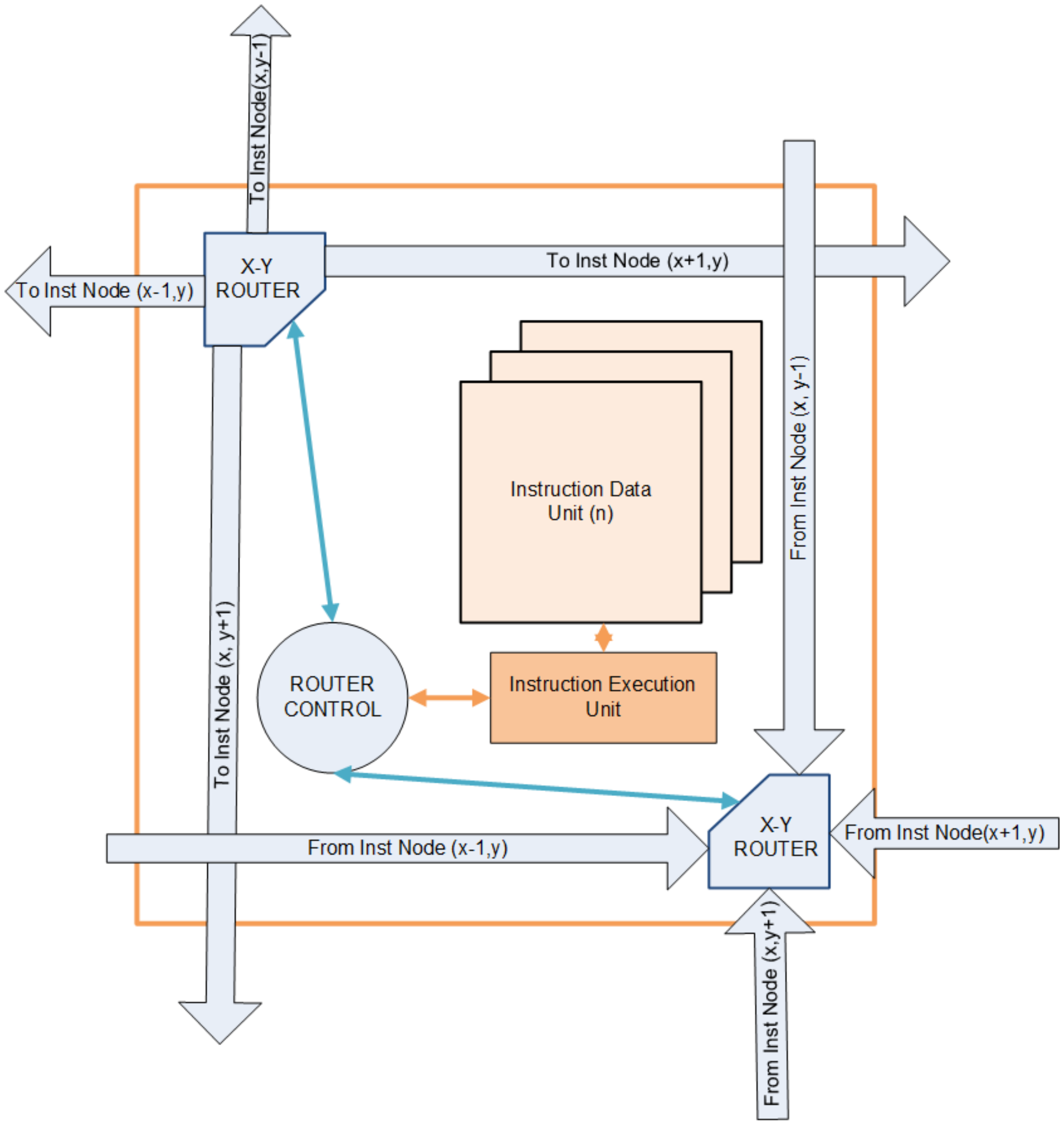


Figure 18 JavaFlow Mesh Network

Memory – General Purpose Processor Interface

Selected elements are connected to rings that interface to the GPP and Memory. Memory ordering is achieved through the MEMORY_TOKEN described in Section 6.3 which is sent on the forward ordered network.

This allows the memory subsystem to receive read and write requests in proper order and to deliver resulting data back to the DataFlow elements. Memory subsystems have received significant research and this design does not place any specific requirements on this subsystem. Figure 19 shows the memory to the side of the DataFlow fabric; however, it could be envisioned in the middle of the chip to achieve shorter paths to the Load/Store Instruction Nodes. Vertical chip stacking could also be used to minimize the delays between the memory and the consuming elements.

Identified by the 'A' in the node in the system diagram, Figure 12, one of the key points of interface between the GPP and the DataFlow Fabric is a series of Anchor Nodes. These are placed like normal Instruction Nodes, but likely closest to the GPP. They form the anchor point for all methods and serve as the instruction load point between the Fabric and the General Purpose Processor. All Serial commands sent to the downstream Instruction Nodes are generated from these Anchor Nodes under distributed control of the General Purpose Processor and the Anchor Node.

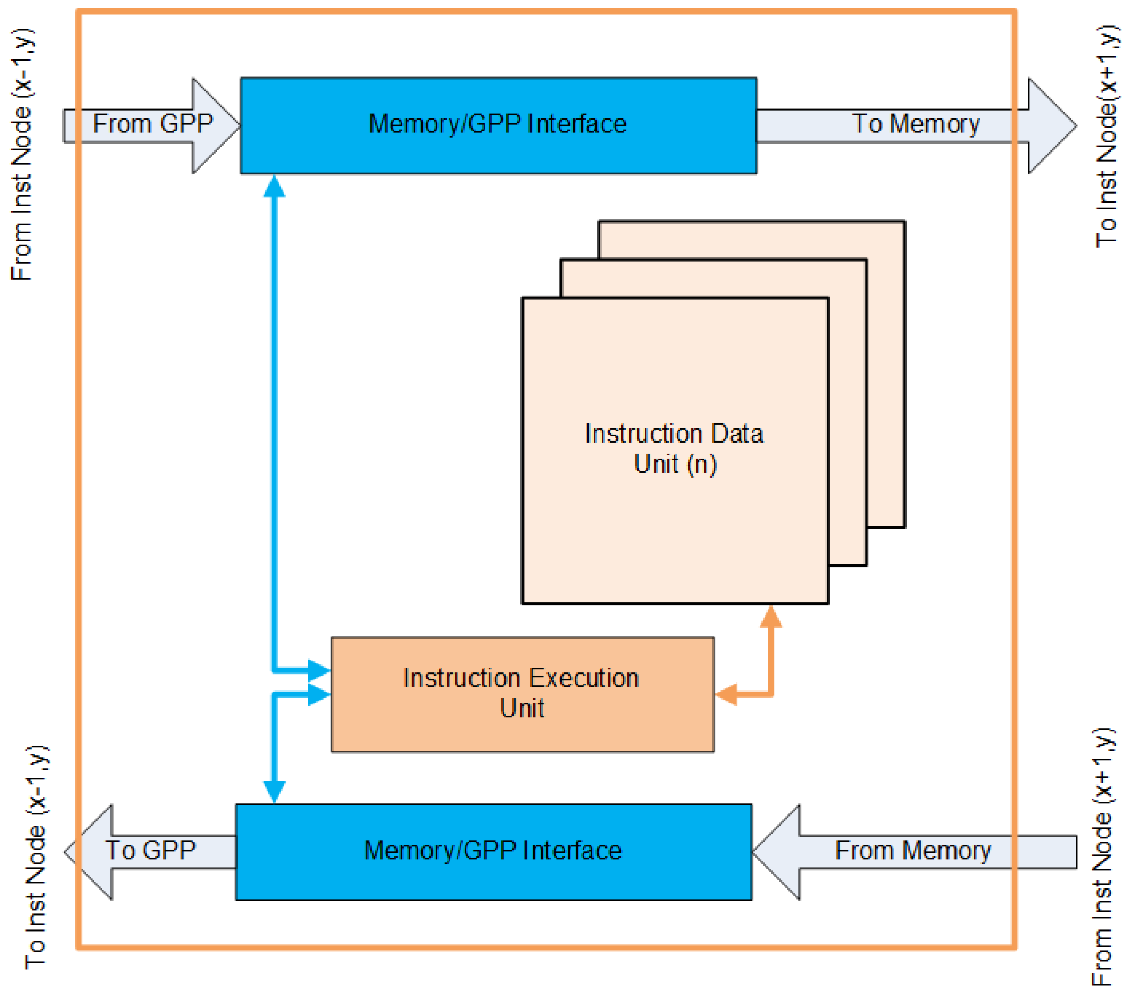


Figure 19 Memory-General Purpose Processor Interface

SECTION 6.2 - DATAFLOW FABRIC MANAGEMENT

Loading a Method

Before a method can be loaded into the DataFlow Fabric, the General Purpose processor must perform the initial functions for all Java classes [5]:

1. Preparation
2. Verification
3. Resolution

There are series of verification steps that must be performed to insure the method complies with the JVM architecture. The resolution step links the symbolic references from Java Class files into references to addresses on either the JVM Method Area or Heap.

One way some JVM's implement this address resolution is through 'Quick' instructions [5]. This process places the actual heap/method area offsets in the method code for the specific instance. The use of the Serial Network and special commands to load new pointers will allow resident methods to be executed for different instances of Class objects without reloading the method.

A fundamental assumption regarding the JavaFlow machine is that since whole methods can be loaded into the machine and kept resident for multiple Object Instances, then the load time is not critical to the overall performance. In contrast, it is assumed that management of the DataFlow fabric and GPP offload is more important, and hence the loading and address resolution process is more relaxed than if the method or a portion of a method had to be loaded for each execution.

Although the translation from control flow instructions to dataflow (Producer-Consumer) instructions will be performed by the DataFlow Fabric, there are 2 steps that can be performed by the General Purpose Processor to relieve some hardware from each node. In addition to the opcode, the DataFlow node will need to know how many data elements the instruction would 'Pop' from the stack and also how many elements it would 'Push' back onto the stack. In the case of all instruction except Calls, this is a direct translation from the opcode. Calls have multiple 'pops' from the stack depending on the signature of the called program. This can be determined from the ByteCode instruction in the Class File and requires analysis of the operand field of the instruction. The values of 'pops' and 'push' for each instruction are described in Appendix A.

Like a JIT (Just in Time) compiler, it is envisioned that a method would be executed interpretively until it becomes clear that it is a candidate for deployment to the DataFlow Fabric. Obviously other strategies can be used to decide which methods are deployed when, and that is beyond the scope of this research.

There are three steps required to load a method and to resolve the addresses from the original ByteCode stream to a Producer/Consumer DataFlow addressing scheme:

- Load the method into the DataFlow Fabric
- Send Source Linear Addresses down the serial network so that Instruction Data Units know the control flow source.
- Send 'Needs Requests' up the serial network from each Instruction Data Node to its source Instruction Data Node so that the Producer/Consumer linkage can be established. (Note that in this case the information provided to the source node is the mesh (x, y, p) address so that DataFlow routing can be performed with the produced data.)

Loading a method from the General Purpose Processor memory is a serial process of retrieving the resolved ByteCode instructions and sending them serially into the DataFlow Fabric. Instructions are the payload of Serial messages with a command of CMD_LOAD_INSTRUCTION. The Anchor node acts as the first interface point between the General Purpose Processor and the Fabric. The Anchor node would likely have a DMA like access to the memory subsystem so that it could retrieve the instructions in a manner consistent with the DataFlow fabric to absorb the data. The only decision that the General Purpose Processor must make is which specific Anchor Node to deploy the method.

The Serial Network accepts each instruction and as it passes through an Instruction Node, a decision is made whether the node is the proper match for the instruction and is the node busy or not. Figure 20 demonstrates this process. The ‘greedy’ allocation process means that a matched non busy node accepts the instruction, marks itself busy and then continues to send subsequent instructions down the network. In Chapter 7, several configurations are proposed including some with all nodes capable of accepting all instructions (homogeneous nodes) and with nodes optimized to the static instruction mix described in Chapter 5 (heterogeneous nodes).

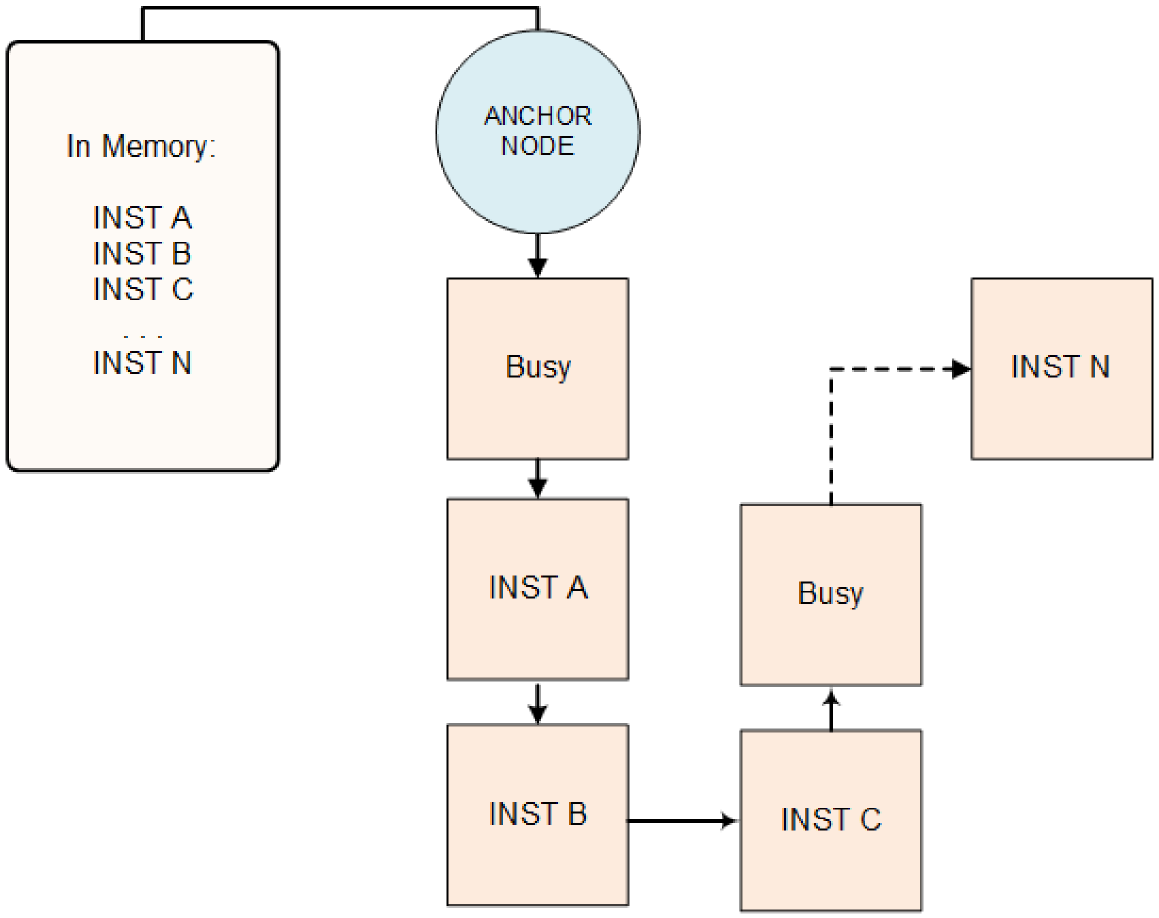


Figure 20 Loading a Method

After the instruction load process, all ByteCode instructions from the method are resident in nodes of the DataFlow Fabric. Instructions know their own linear address from the original stream, and the control flow modification instructions know the target linear address of the taken path. Since the Serial Network can be used to send messages to just the adjacent node, only those nodes that are non-sequential must be explicitly identified. All other instructions implicitly proceed to the next sequential linear address. The Anchor Node is made aware of the completion of this process by the bottom instruction in the method passing a TAIL_TOKEN message back up the Serial Network to be received by the Anchor.

DataFlow Address Resolution

The next function in the loading process is to perform the translation from the control flow oriented ByteCode instructions to a DataFlow Producer-Consumer system. Although the descriptions will indicate actions performed by an Instruction Data Unit, in all cases the Instruction Execution Unit performs these function on behalf of each Instruction Data Unit.

The first step of this function is for each Instruction Data Unit with a non-adjacent subsequent Instruction Data Unit to generate and send a message down the Serial Network identifying itself. Since this is exclusively performed on the Serial Network, mesh addresses are not necessary at this point. The process is initiated from the Anchor Node (GPP) via a CMD_SEND_ADDRESSES_DOWN message.

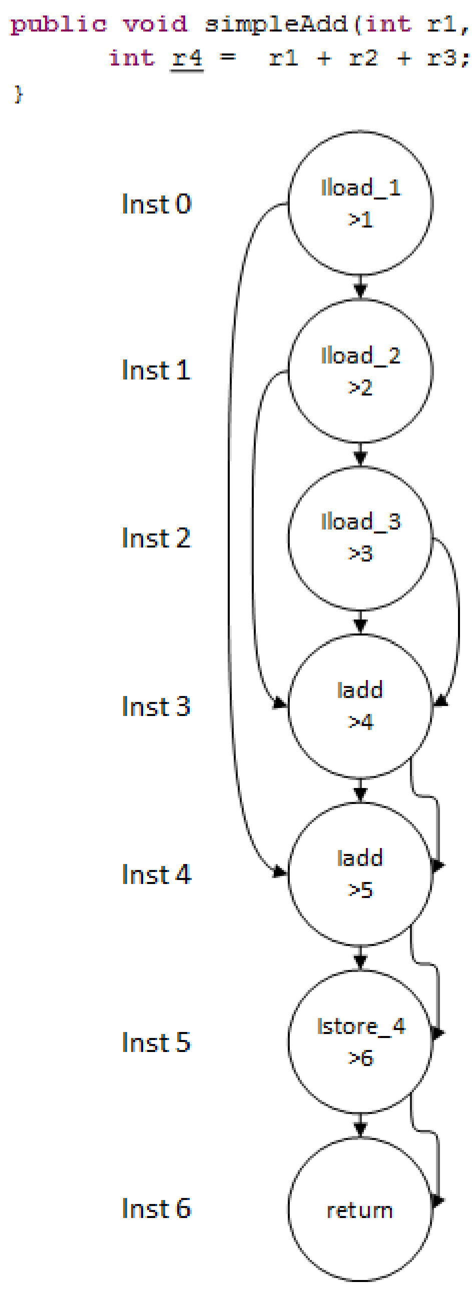
Note that since some targets can be behind the existing instruction, either the Up Serial Network could be used or the Network must wrap at the bottom instruction. In line with the execution strategy to be described in the next section, the simulation of these two processes use the CMD_SEND ADDRESSES_DOWN followed by a TAIL_TOKEN and monitor the receipt of the TAIL_TOKEN back to the Anchor node indicating a complete cycle of all instructions.

At this point in the process each Instruction Data Unit has an awareness that it has at least one upstream source instruction and it also has explicitly received the address of any jump or GoTo instructions that would transfer control to it.

Step two of the function now begins with the Anchor node sending the CMD_SEND_NEEDS_UP. This command tells an Instruction Data Unit to generate as many messages as the ‘Pop’ value of the instruction and send messages up the Serial Network. The critical factor in this process is that each instruction must send its messages up before propagating any messages from below. This might stall the Serial Network or require buffering at each node for these messages. Section 7.2 shows the average and maximum buffer queues needed in the simulation of this process. In most cases the ‘toAddr’ field in the Serial message would a special value of ‘Previous Instruction.’ The payload of these messages is the Mesh Address of the sending instruction. This is the process of translating the stack ‘Pops’ in the normal ByteCode execution to Producer DataFlow addresses in the upstream instructions. For nodes with ‘Pop’>1, the ‘Side’ value is inserted into the Serial message so that the Producer knows which side of target to address messages after execution.

If the upstream node has a ‘Push’ value, then the instruction captures the message and uses the Mesh Address and Side for its target address. If the upstream instruction has no ‘Push’ value or if the ‘Push’ value had been previously satisfied, then the message is resent to the next upstream node.

Figure 21 provides an example of a very simple program and how the address resolution process may be accomplished. The simple method receives 3 register values and then adds them and places the result into register 4. In this example there are no forward or backward jumps so all serial addressing is done using the ‘nextAddress’ or ‘previousAddress’ special destination codes.



```
public void simpleAdd(int r1, int r2, int r3){
      int r4 =  r1 + r2 + r3;
}
```

Figure 21 Simple Address Resolution Example

The first three iLoad ByteCode instructions load data from the corresponding register number and will place it into the DataFlow Fabric. The iAdd instruction will pop

2 values and push one result. The iStore instruction will pop one value and push none. The return pops zero (void) and transfers back to the calling method.

When the command CMD_SEND_NEEDS_UP is received by each of the example Instruction Data Units, a message is sent to the immediate upstream node for each pop required. Instruction #5 which requires 1 pop will send a message to Instruction #4 who produces 1 push. Based on this match, the curved link between the Instruction Data Units #5 and #6 is created. Instruction #4 requires 2 pops and therefore sends 2 messages up the network. The first message is seen by Instruction #3 whose push value is 1 and therefore the linkage between #3 and #4 is established. When the second message from #4 is received by instruction #3, the push has already been satisfied. Therefore the message is forwarded to the source for #3, which is instruction #2. Since all messages must be sent before any are transferred, instruction #2's single push had been resolved with a link to instruction #3. Therefore this second message from #4 must be sent further up the chain until an open push is found. In this example, instruction #0 is finally found to have an open push, and the linkage from #0 to #4 is established. While this example is trivial, it demonstrates the key elements of the resolution process. The process becomes more complicated with jumps back (loops) and DataFlow merges where Instruction Data Units may have multiple source nodes. This requires a number of messages equal to the product of the pops and the number of source nodes.

In the case of a DataFlow merge where an instruction has 2 or more upstream nodes, then an additional complexity must be handled. As part of the message payload or possibly another portion of the Serial message, there must be a 'Branch ID' tag. The results shows this need be only a single bit in the analyzed methods, but the multiple sources must be both sent messages and also those messages must be tagged indicating that they multiple messages from a single node. As long as the paths are separate the 'Branch ID' tags are used to link addresses. These can pass through normal instructions and through ‘GoTo’ instructions. However when a jump instruction which is a control flow split, then only the Branch ID==0 tag is propagated. This is based on the Java Virtual Machine restriction that all paths to any instruction must have the same stack configuration.

Like the forward address resolution process, this should continue for a full cycle around all the instructions. Analysis has shown that there are no back merges. That means all dataflow references are forward in the existing data and according to JVM restrictions. Checking for this might require a full cycle however.

At the end of this process each instruction which has any ‘Push’ value, has one or more Mesh Addresses as the consumer for the data. Note that unlike other machines, these ‘Push’ addresses are generated automatically and not part of the instruction set stored in the General Purpose Processor’s memory. This allows multiple fan out targets depending on the resources of the Instruction Node. The DataFlow Results section demonstrates these values.

There are 2 validation measures that can be performed to insure the initial ByteCode instruction stream was valid. If any instruction with a non-zero ‘Push’ value has less DataFlow targets than the ‘Push’ value then an error can be logged. Similarly if the top instruction sends any requests for linkage to the Anchor node, another error could be logged.

```
......
(0),32,33,,iload, >>43.M-.2.0; <<pop=0;push=1;lr
(0),33,34,,iload, >>35.--.2.0; <<pop=0;push=1;lr
(0),34,35,,ldc, >>35.--.1.0; <<pop=0;push=1;cn
(0),35,36,,ixor, >>39.--.2.0; <<pop=2;push=1;,ai
(0),36,37,,iload, >>38.--.2.0; <<pop=0;push=1;lr
(0),37,38,,ldc, >>38.--.1.0; <<pop=0;push=1;cn
(0),38,39,,ixor, >>39.--.1.0; <<pop=2;push=1;ai
(+),39,40,42,if_icmple, <<pop=2;push=0;jp
(0),40,41,,iconst_1, >>43.M-.1.1; <<pop=0;push=1;,mv
(+),41,,43,goto, <<pop=0;push=0;gt
(0),42,43,,iconst_0, >>43.M-.1.0; <<pop=0;push=1;mv
(0),43,44,,iadd, >>44.--.1.0; <<pop=2;push=1;ai
(0),44,45,,istore, <<pop=1;push=0;lw
......
```

Straight lines – Control Flow
Curved lines – DataFlow

Class: gnu\java\Math\MPN
Method (snippet) sub_n([I[I[II)I

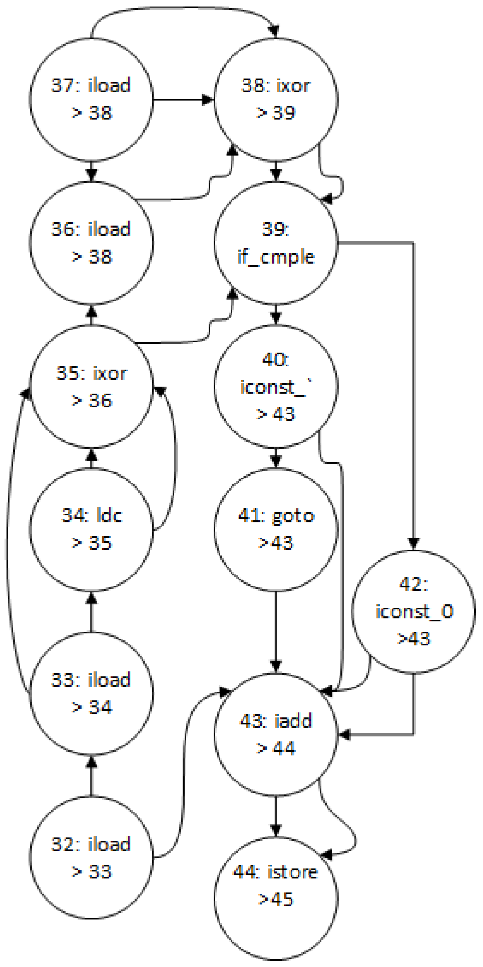


Figure 22 DataFlow Address Resolution

Figure 22 demonstrates a more complex example of a code segment from the method 'gnu\java\math\MPN\sub_n([I[I[II)I.' Linear instructions 32 through 44 are shown with the original ByteCode inset and the resolved DataFlow addresses inside the instructions. This example shows the effect of a dataflow merge as both instructions 40 and 42 push data to side one of instruction 43. Note that instruction 32 pushes a value for side 2 of instruction 43 as part of the DataFlow merge.

Given that the JAVAC compiler uses a strategy to maintain all interblock communications with registers, it is hard to imagine a scenario when a valid merge back would occur. If it is discovered, the method would have to be excluded from the DataFlow Fabric or a design enhancement to translate this stack transfer into a pseudo register. The complexity of this latter process is inconsistent with the minimalist strategy for the JavaFlow machine.

For completeness, the following describes the values shown for each instruction in the list in Figure 22:

- (x) Control direction where (0) is normal next instruction, (-) is a possible jump back, and (+) is a jump forward.
- A1, A2, A3. A1 is the serial address of the current instruction, A2 is the next instruction (not taken), and A3 is the next instruction (taken)
- >> A4, mb,s,i << For each push, A4 is the destination serial address, 'm' is a flag with 'M' indicating a dataflow merge at the destination; 'b' is a flag with 'B' indicating a back merge; 's' is the side of the destination node; and 'i' is the Branch ID tag.
- 'Pop' and 'push' values for the current instruction
- Abbreviated instruction group

Initialization and Execution Start

After the method has been loaded and resolved, it is ready for execution. Each node can enter the STATUS_READY state and await serial and mesh messages as part of the execution process.

Management and Cleanup

While the General Purpose Processor is not involved in the actual assignment of instructions to specific nodes, it obviously has to have some idea about how many methods are deployed and how they are being utilized. As mentioned in the initial section on Java, one of its key advantages is JVM management of all machine resources such as both the Method Area and the Heap. As Classes become de-referenced, then it is possible to remove methods from the DataFlow Fabric. Assuming that the Class would not be re-loaded, the GPP can issue the CMD_UNLOAD_INSTRUCTION to the Anchor node and this serial message would propagate to all previously loaded instructions and make them free for use in another method.

With multiple Anchor Nodes methods can be reasonably distributed across the DataFlow Fabric and with the objective that the number of spanned nodes is in line with the original measurements presented in Chapter 7. An observation point on this is that unused Instruction Nodes are not necessarily a detraction from overall machine operation. One of the objectives of the JavaFlow machine is to manage the power dissipation of the system, and to that end nodes not housing instructions can have sections of the node powered down to conserve power. Note that only the serial and mesh routers must be active on all nodes and then only consume power when messages are routed through them. In each Instruction Node, the instruction execution unit and any instruction data units can be powered off if no instruction is loaded. This concept is a radical departure from traditional processor performance optimizations. Note that the serial network router and mesh network routers must remain powered due to their simplistic design, unless whole contiguous sections of the chip are powered down.

SECTION 6.3 - METHOD EXECUTION

The key strategy for executing the previously loaded instructions is to start a bundle of tokens in the form of messages down the Serial Network. The first element in this bundle is the HEAD_TOKEN. The HEAD_TOKEN is the 'rabbit' which leads the way and is the primary translator between dataflow processing and control flow ordering. This token proceeds as fast as possible through all nodes in the network until a possible control flow change is encountered. Depending on the direction of the possible control flow change, and when the decision is made, the token either proceeds to the target instruction or is buffered at the jump or GoTo node.

The second token is the MEMORY_TOKEN which is used to achieve memory ordering. The payload of this token is a sequential memory order number which is incremented for each ordered memory operation and then handled by the memory subsystem.

The next tokens are REGISTER_TOKEN types. These tokens contain the register number along with the register data. Note that the invocation of a method has the calling method place its parameters in the local registers of the called method. In the case of methods that are class instances, register 0 is set to the pointer to the instance data in the heap.

Finally the TAIL_TOKEN ends the token bundle. This TAIL_TOKEN may never pass any other token, and it is often used as a barrier to handle branch backs in the method execution.

Figure 23 shows this bundle as it is ready to propagate through a method.

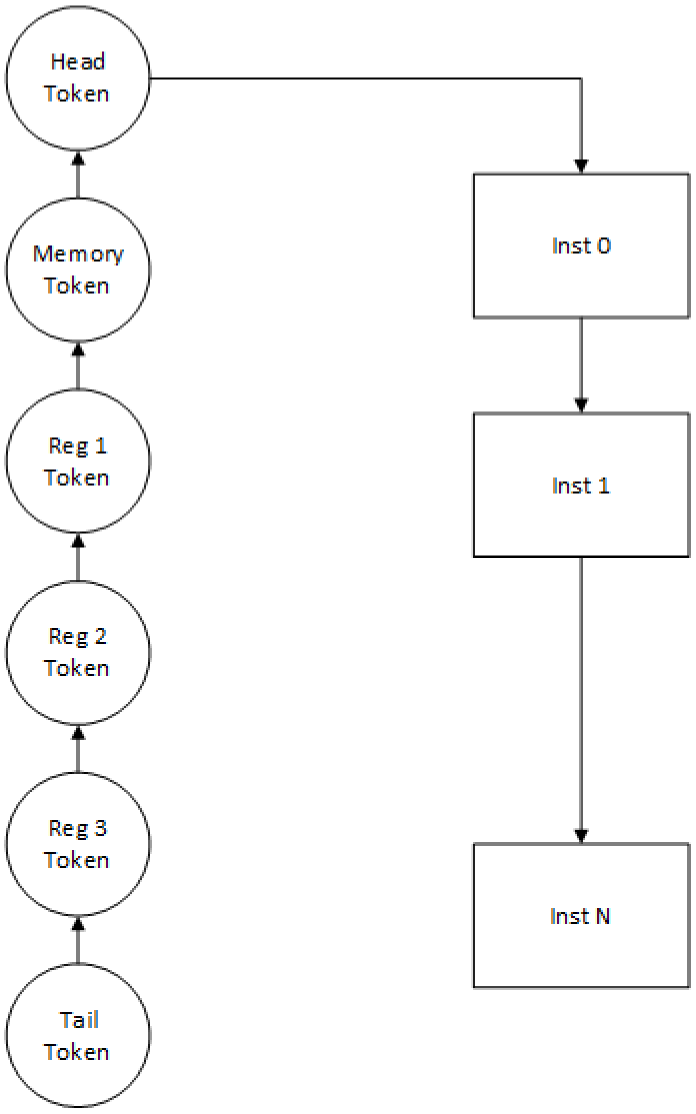


Figure 23 Token Bundle

Instruction Group – Arithmetic/Logical/Move Operations

The first type of operations to be described are the Arithmetic/Logical/Move operations which consist of a significant number of the overall set of ByteCode instructions. Note that while each opcode performs significantly different operations, the way these work with respect to both the Serial and Mesh networks is the same.

Once one of these instructions is made ready through loading or the completion of a loop (which is described in the Control Flow Instructions), the only condition for the execution or 'firing' is the receipt of the HEAD_TOKEN and the number of operands identified by the opcode. When the HEAD_TOKEN arrives, the status of the instruction is set to 'headTokenReceived.' All serial tokens except the TAIL_TOKEN are unconditionally passed down the Serial Network. The TAIL_TOKEN is only passed after the instruction fires.

The criteria for firing is that the number of received mesh messages matches the number of 'Pops' expected by the instruction. Note that many of these instructions have a 'pop' value of zero which means the instruction can fire upon receipt of the HEAD_TOKEN.

Mesh Messages bringing the number of 'Pops' to the node may occur at any time. When they arrive, they are stored in the specified 'side' of the instruction and the 'PopsReceived' counter is incremented. If the 'PopsReceived' counter reaches the 'Pop' value of the instruction, and the instruction has not fired, the instruction continues to wait for the HEAD_TOKEN so that both criteria are available for firing.

When the instruction executes or 'fires' the processing is performed on the input data and the result data is then inserted into one or more Mesh Network messages for routing to consumer nodes in the network. As described earlier, the array of destination addresses is limited only by the buffering in each node as opposed to limitations on the architectural definition of the instruction layout.

When the TAIL_TOKEN arrives, it is passed down the Serial Network if the instruction has fired. If the instruction is still awaiting Mesh Data, then the TAIL_TOKEN is held at the Instruction Node until it fires. This is to insure that the Token Bundle represents the entire span of execution of the current method. Note that all Instruction

Nodes must be capable of buffering or recreating the TAIL_TOKEN message for subsequent passing after the node fires. Control flow nodes must buffer more tokens.

Instruction Group – Register Operations

Register operations in the ByteCode instruction set are responsible for transferring data between the local registers and the stack which in the JavaFlow machine is the arcs of the DataFlow Fabric. Local Read instructions take register data and send it to Mesh Node consumers. Local Writes take data from the Mesh Node and transfer to the Serial Network overwriting the value of register data. There is one special register instruction that simply modifies a register by incrementing the value and re-sending down the Serial Network. Like the all instructions, the HEAD_TOKEN must arrive before execution. Also the TAIL_TOKEN cannot be propagated until the instruction fires.

Local Read Instructions

For Local Reads, by definition the HEAD_TOKEN arrives before any register tokens. The instruction logic must compare the register number from each subsequent REGISTER_TOKEN with the register required. When a match in the REGISTER_TOKEN and the operand is achieved, then the state becomes 'myRegisterTokenReceived' and the Local Read instruction can fire.

In the case of the Local Read, the firing takes two actions:

- Like the Arithmetic instructions, one or more Mesh Messages are generated to send the register data to downstream nodes.
- Since the data in the register is unaltered and is likely to be used subsequently, the REGISTER_TOKEN must also be sent down the serial network.

Since the only requirement for ‘firing’ is the receipt of the specific REGISTER_TOKEN the REGISTER_TOKENS are never re-ordered as a result of a Local Read Operation.

Local Writes

The Local Write operations are slightly more complicated in that two conditions must be met for firing

- The receipt of the required number of Mesh Messages indicated by the ‘Pop’ value in the instruction. Note this could be 0 if an immediate operand value is being transferred to the register or 1 if stack data is being written to the register.
- The receipt of the HEAD_TOKEN.

Since these events can occur in any order, the instruction must be able to buffer the data from the Mesh Message. Note that this instruction ‘kills’ the value of a register and hence does not have to save or propagate the register value. This can result in the re-ordering of the REGISTER_TOKEN messages.

The result of this instruction ‘firing’ is a Serial Network message with the updated register data send down the network.

Local Increment

The Local Increment instruction is a combination of the Read and Write local instructions. No Mesh Network data is involved, so the only condition for firing is that the REGISTER_TOKEN with the specified register arrive at the node. When the instruction fires, the value of the operand is added to the value of the register and the REGISTER_TOKEN send down the Serial Network. This instruction is used for loop counting in methods.

Instruction Group – Storage Operations

JavaFlow storage instructions combine data from messages internal to the DataFlow Fabric with messages to and from the Storage Subsystem. The Storage Subsystem is reached through a ring network which to which each Storage node interfaces. Note that this interfacing becomes more challenging in the case of homogeneous nodes where every node must interface the Storage rings.

If a homogenous node assignment approach is required for better node management, then still, only selected nodes would have the interface to the Storage Subsystem and, messages intended for Storage would have two hops: one from a homogeneous Instruction Node implementing a storage operation, to a special node with an interface to the Storage rings, and then the message to and from the memory system.

There are three types of read and write storage operations performed by ByteCode Instructions:

- Unordered constant access to the Method Area. Specific op codes are defined to access constants, and these are loaded before the method is deployed and are not modified during execution. This allows unordered access to these constants.
- Ordered access to the Method Area for Class data (static)
- Ordered access to the Heap Area for Object (instantiated Class) data

The memory subsystem is responsible for the ordering of memory operations, and the DataFlow fabric utilizes the MEMORY_TOKEN to facilitate this strict ordering system. In normal Instructions the MEMORY_TOKEN is simply passed along like the other tokens. With ordered Storage instructions, this MEMORY_TOKEN is first incremented and then the new value used along with the address/data to send to the memory subsystem. While WaveScalar [35] uses a particularly elegant system for memory ordering, the JavaFlow approach is consistent with the minimalist design and does maintain strict memory order.

With the exception of the variants discussed above, the memory access instructions work similar to other instructions in JavaFlow. The HEAD_TOKEN must arrive before the instruction fires. The number of 'Pops' must be received from Mesh Network, and when 'Pops' equals 'PopsReceived' and the HEAD_TOKEN is present, then the instruction can initiate firing. Unlike other instructions a message is first generated to the memory subsystem and the Instruction can enter the state of 'waitingForService'

For memory reads, obviously the node must remain in the 'waitingForService' state until the memory system returns the result. At that time the appropriate number of Mesh Messages are generated to send the resulting data do downstream nodes. The instruction is not considered 'fired' until the Service message is received.

For memory writes, JavaFlow processing continues and is only stalled if another memory read operation is encountered. With the memory subsystem having the MEMORY_TOKEN order tags, it can manage the ordering of memory accesses. The write instruction is considered 'fired' when the service message is sent to the memory subsystem.

As in all instructions, the TAIL_TOKEN must be held until the instruction has completely fired.

Instruction Group – Service Operations

Service instructions are very similar to Memory Read/Write instructions except the Service message is sent to the General Purpose Processor. Examples of Service instructions are instantiations of new Objects or Arrays and tests of properties of these objects. There are some instructions that are both very infrequently executed and also inconsistent with the minimalist design approach of JavaFlow and are delegated to the General purpose processor.

Depending on the nature of the memory subsystem, the MEMORY_TOKEN value may have to be sent to the General Purpose Processor to insure that no memory ordering is violated between the General Purpose Processor and the DataFlow Fabric.

Instruction Group – Control Flow Operations

The coarse group of instructions that modify the control flow of the program are the most complex to implement in this hybrid DataFlow machine. This group of instructions include many different groups of operations:

- GoTo
- Conditional Jumps
- Calls (invoke)
- Returns

Like the instructions described above, these must have the HEAD_TOKEN and the 'PopsReceived'=='pop' in order to fire. However since these instruction modify the control flow of the program there are strict conditions for passing tokens down the Serial Network.

- Before the instruction fires, all tokens must be buffered and not passed along the Serial Network
- When the instruction fires, and the next address is forward (i.e. linearAddressThis < linearAddressTarget) then tokens can be routed to the target next address. Note that in this case the messages on the Serial Network will have an explicit address which is ignored by all intervening

nodes. Upon receipt by the target node, the tokens are then processed as usual.

- When the instruction fires, and the next address is backwards (i.e., linearAddressThis > linearAddressTarget) then all tokens must still be buffered until the TAIL_TOKEN arrives. At that time the Upstream Serial Network is used to route all tokens to the target Instruction Node. As the HEAD_TOKEN passes up the Serial Network, each instruction from the same thread/class/method must also reset to the 'stateReady' in case any data had been received by a node that was not fired in the previous loop.

The reason for this stall in the execution of the instructions is to allow any processing in the loop to complete before the loop is re-executed. This buffering and instruction stall is the key element of allowing complete methods to be resident in the DataFlow Fabric and avoids the pressure on the instruction fetch subsystem where only partial methods can be loaded. .

While each group of the control flow instruction is basically the same, there are some differences:

- The GoTo instruction is an unconditional jump either forward or backward. If the target is forward, then this instruction can fire immediately upon receipt of the HEAD_TOKEN. If backwards, then the TAIL_TOKEN is required to fire.
- Jumps pop 1 or 2 values for comparisons and have 2 options for the next address. If not-taken, then instruction processing resumes at the next linear address. If taken, instruction processing resumes at the identified target linear address.

- Return. These are a series of instructions that end the execution of the current method. To insure that all processing is complete, the Returns must function like a back jump where the TAIL_TOKEN is necessary for firing. The data that would be sent to another Mesh Network node is passed back to the GPP where it is then placed on the stack of the calling method. Note the enhancement option is described in Section 6.4 where calls and returns could be handled internal to the DataFlow Fabric.
- Call (invoke). These are a series of instructions that transfer control to another method and resume execution when that method completes. While the JVM specification is not precise, and since the subsequent instruction is always forward, most tokens can be passed while this instruction is executing. The instruction may fire when the HEAD_TOKEN is received if 'pop'==0 or when the 'PopsReceived'=='pop.' Note, these instructions can have high 'pop' values as large number of parameters can be passed during the call process. Since another method is being executed, the MEMORY_TOKEN must be sent to the new method, or at least buffer this token until the other method returns. This would be required to preserve memory ordering between methods. The receipt of the TAIL_TOKEN is then used to dequeue any buffered tokens. (For the simulation, only the TAIL_TOKEN is buffered until the instruction fires.)
- Special Instructions. There are several instructions that can affect the control flow of the method which are not implemented in the simulation due to lack of usage in the benchmark methods. These instructions are discussed in the next paragraph.

Instruction Group – Special Instructions

In addition to the service instructions that are implemented in the General Purpose Processor, the ByteCode instruction set has a series of special operations whose frequency is sufficiently small to be ignored in this simulation, but is discussed briefly in this section.

To assist in the implementation of the 'Finally' clause in the Java language, a 'Jump Subroutine (jsr)' and 'Return from Subroutine (ret)' have been defined. The benchmarks had little to no usage of these instructions, however there are several options for their implementation should it be necessary. Rather than utilize the GPP to handle this, another token could be created (RETURN_TOKEN) whereby the 'jsr' instruction simply writes its linearAddress +1 into the token which then passes down the Serial Network. Any 'ret' instruction would then capture this token and when fired would act like a jump to the address presented in the RETURN_TOKEN.

The 'Lookup Switch' and 'TableSwitch' instructions are control flow modification instructions implemented the equivalent of Java or C 'switch' structures. A key value is used to fire the instruction and then based on a series of comparisons, one of multiple next instructions is invoked as a jump. Implementing these function in the DataFlow fabric is counter to the minimalist strategy being used for JavaFlow. The GPP might assist in execution, or methods with these instructions might be ignored for execution in the DataFlow Fabric as was done in the simulation results. The data in Chapter 5 indicates a minimal impact for this decision.

Anchor Node

Anchor nodes shown in Figure 12 and Figure 20 are Instruction Nodes like the Control Flow Operation nodes, except that they form the primary interface between the loading of the method into the DataFlow Fabric and the General Purpose Processor. These nodes for the head of the Serial Network for each method and could be completely responsible for the commands sent down the network to resolve the DataFlow addresses. These nodes would also maintain the status of a deployed method so that if a different thread in the GPP or the DataFlow Fabric attempted to execute the method, the proper busy/available signal could be returned.

Exceptions

The Java Virtual Machine has a wide range of exception conditions defined, and many of these must be detected by the DataFlow Fabric and the Instruction Nodes. Obviously each node must report arithmetic exceptions by halting operations and sending messages to the GPP for handling. The tag structure described in Section 3.6 called for hashed representations of the object signatures in storage operations to insure that no improper casting/aliasing was being performed.

The handling of array bounds checking is an example of how the balance of the minimalist design of the Instruction Nodes can be balanced with the opportunity of parallelism of memory accesses. As part of the address resolution process, those instructions involved with accessing Java array elements would access the array boundary information from the Constant Pool in addition to the actual pointer to the array. In this way, when the instruction is executed at a later time, the array bounds are available to create an exception if they are being violated by the runtime method.

In general the strategy for handling exceptions is to use the DataFlow Fabric when necessary to detect errors and always rely upon the General Purpose Processor to handle the exception conditions. Usually the method will be terminated and with execution resumed at another method or at a point in the current method which would be a 'Finally' clause. The handling of this construct was discussed in the section on Special Instructions.

SECTION 6.4 - ENHANCEMENTS

In addition to the described functions of the Java DataFlow machine, there are several enhancements that could be considered for is implementation.

Predication can be used to not only enable instructions following branches, but can also be used to allow speculative execution. Although not currently envisioned as part of the distributed intelligence in the current DataFlow Fabric, being able to fire multiple target instructions and hold the storing (or rolling back) data could be a design enhancement to increase performance. This method of performance enhancement does come at the price of increased and wasted power consumption.

Although complete methods are deployed to the fabric and all address and operand resolution is done by the distributed intelligence, inter method communication is still performed by the GPP. By deploying additional information about called methods, the ByteCode instructions invoking other methods could directly transfer the register values to another method resident in the Fabric. Implementing this capability could reduce the call time significantly but additional communication would be required to insure that only one thread/class/method is executing in a method at one time. Therefore even if the called method is resident in the DataFlow Fabric, a message requesting service would have to be sent by the calling node prior to sending any parameters. If the called anchor node is not busy, then it would be marked busy, and an ACK message send back to the calling method which would then send parameters. Upon completion, the called routine would execute its Return instruction and a Mesh Message with the returned data passed back to the calling method.

Since many of the JVM ByteCode instructions simply move data in the stack or in the registers, there is the opportunity to eliminate instructions by having a node declare itself void. This is the 'folding' process referred to in previous machines and could be performed automatically via the distributed DataFlow Fabric. This additional function would be performed after the linkage process described in Section 6.2. Nodes that perform only data transfers would send messages up to their producer nodes to change the producer node targets to the targets of the redundant nodes. The redundant nodes could then be returned to the unoccupied state for use by another method.

Obviously compiler enhancements could be used to create more efficient code. Traditional compiler techniques could optimize the code while replacing calls to small methods simply getting and setting fields with get and put instructions would reduce the overall pressure on method calls with associated overhead.

An opportunity exists with the parallelism of the JavaFlow Data Fabric to assist in the area of Garbage Collection. Some implementations utilize an additional level of indirection to access both Heap and Method area data [5]. While the details of these implementations are dependent on the overall JVM system, the objective is to insert an additional level of indirection so that a single pointer can reference for all of the data for an instance so that it can be moved easily during Garbage Collection. The parallelism and communications capability of the JavaFlow machine has the ability to quiese the execution of a method and pass serial commands through the Serial Network to force re-calculation of memory pointers as a result of Garbage Collection.

Network enhancements can be employed to improve performance. The effect of serial clocking is measured in Chapter 7 and clearly impacts performance. More parallelism in the serial networks or enhance routing techniques could improve the system towards the case where the delay of the serial network was removed. Similarly since the size of the data transfer arc between Producer and Consumer nodes is small, there may be network configurations optimizing small arc data transfers that could improve performance.

Further enhancements to the serial network could be applied in the area of InstanceID matching. Since this matching is key to the performance of the network, some level of network segmentation and hashing might get the number of matching elements to a very small number of bits. If only one copy of a method were to be allowed to be loaded at a time, then further reduction in the size of the InstanceID could be realized.

An obvious area of enhancement would be in the memory subsystem and the ability of Instruction Nodes deep inside the DataFlow Fabric to present memory requests quickly. While a ring network is proposed for selected Instruction Nodes, this system would seem to be a great candidate for a 3D memory system where memory requests could be presented to the subsystem vertically from within the Fabric.

SECTION 6.5 - SUMMARY

The JavaFlow machine has been described to a level to allow a simulation to demonstrate the results in the following chapter and to insure viability. All Java ByteCode instructions are addressed, and all affecting the performance of the benchmarks have been precisely defined. Special case instructions and situations have been described along with options for their implementation. Precise bus sizes and tag encoding are not defined at this time, as significant silicon level optimizations are possible in that area. A series of enhancements are described which can offer even further levels of performance enhancements.

Chapter 7: Results

SECTION 7.1 - RESULTS OVERVIEW

In the previous description chapters, a series of qualitative advantages for the JavaFlow system have been described. These include design simplification, automatic management of the DataFlow Fabric, power reduction through unused nodes, and the ability to handle many threads simultaneously. A quantitative measurement process must be defined to establish a performance figure of merit so the various system configurations can be compared and then matched against alternate implementations. A key aspect of this process is to neutralize the effects of technology on these measurements along with minimizing dependencies on some key design trade-offs that are beyond the scope of this current research. Two examples of this latter point include the configuration and performance of the memory subsystem and the exact performance of the individual processing nodes.

This Results section is broken into two segments: DataFlow Analysis and Performance. The DataFlow Analysis is the results of performing the loading and resolving the dataflow producer/consumer addresses described in the previous chapter. The Performance Analysis is the results of simulation analysis against a defined baseline configuration.

Note that even though the interactions of all the nodes in the system is described as GALS (Globally Asynchronous, Locally Synchronous [48, 49]), the measurement strategy still implies a clocking structure for both the Mesh and Serial Networks. The simplicity and asynchronous nature of the Serial Network supports the assertion that multiple serial messages can be transferred between a single Mesh Cycle. The Mesh Cycle is envisioned for performance analysis to be based on both the processing function inside the node and the router functions transferring data between nodes.

SECTION 7.2 - DATAFLOW ANALYSIS

Although initial results of the dataflow analysis was included in the Chapter 5, Benchmarks, this section summarizes these results. This data was obtained as the methods were loaded into the simulated environment, and the resolution process described in Section 6.2 was completed.

With a filter (Filter 1) restricting the size of the methods to between 10 and 1000 instructions, Table 9 shows the sizes of the methods, registers and stack. With a median of 29 instructions, the JavaFlow DataFlow Fabric could house many methods in a 1000 to 10,000 node system. The median and average number of stack elements and local registers appear reasonable for nodes described in Chapter 5. Those methods with the maximum number of registers are likely to be excluded from the DataFlow Fabric due to buffer limitations on each node, however,

Table 9 General Data Flow Analysis – Filter 1

	Static Inst	Local Regs	Stack	Back Merge
Mean	56	4.45	3.88	0
Std Dev	86	3.42	1.70	0
Median	29	3.00	4.00	0
Max	931	31.00	14	0
Min	10	0.00	1.00	0

Table 10 demonstrates the characteristics of the actual DataFlow execution by analyzing the FanOut and Arc sizes. The FanOut is the number of consumer nodes to which a producer node sends data. Due to the lack of optimization in the JAVAC compiler, these numbers are very small. While the ability to support larger FanOut than in the TRIPS machine is a feature of JavaFlow, this data indicates that this feature might have more use with a higher level of compiler optimization.

The Arc is the linear length of the data transfer from the producer to the data consumer. The Arc Average is the average for all arcs in the method while the Arc Max column only includes the maximum arc in each method.

This data led the design assumption towards a 10 wide node structure as a segment of the DataFlow Fabric. The goal is to compress the linear method into x-y coordinates that minimize the overall arch length when using the DataFlow fabric.

Table 10 DataFlow FanOut and Arc Analysis - Filter 1

	FanOut Avg.	FanOut Max	Arc Avg.	Arc Max
Mean	1.04	1.53	1.88	6.88
Std Dev	0.07	0.65	0.68	7.69
Median	1.00	1.00	1.70	5.00
Max	1.40	4.00	7.20	187.00
Min	1.00	1.00	1.00	1.0

Table 11 shows the size of queues necessary to resolve addresses from the original ByteCode structure to the DataFlow Producer/Consumer addressing. Section 6.2 described the process of sending messages up the Serial Network in order to establish the proper linkages. The key aspect of this process is that each node must send all its requests upwards before processing or sending any received requests from below. This implies a level of buffering at each node. The mean and median values appear reasonable, although some methods may have to abort due to an overflow in buffer requirements.

Table 11 DataFlow Resolution Queue Analysis – Filter 1

	Max Q Up
Mean	3.03
Std Dev	1.36
Median	3.00
Max	11.00
Min	1.00

Table 12 shows the analysis of DataFlow merges. In the context of the JavaFlow machine a DataFlow merge occurs when a single Instruction Node has multiple source nodes each passing a data element. Note that this occurs infrequently in methods and is the driving force for the buffering described in Table 11.

Table 12 DataFlow Merge Analysis - Filter 1

	Merges
Mean	0.29
Std Dev	0.93
Median	0.00
Max	9.00

Table 13 shows the analysis of forward jumps in the set of methods in Filter 1. These are control flow jumps and therefore do not require a complete stall on the Serial Network. The average length of these jumps shows that there may be an opportunity to utilize the Mesh Network to transfer this data rather than the Serial Network. For the simulated results where variable speeds are used in the Serial Network, this option was not utilized.

Table 13 DataFlow Jump Forward Analysis - Filter 1

	Forward Jumps	Avg. Length	Max Length
Mean	3.07	12.04	22.48
Std Dev	4.94	24.91	54.48
Median	2.00	7.00	7.00
Max	58.00	175.00	803.00

Table 14 shows a similar analysis for backward jumps. As described in Section 6.3, until the direction of the jump is confirmed and then always if backward, all Serial Network Tokens must be buffered at the jump instruction. This could be damaging to the performance of the system, but it is noted the number of back jumps is significantly smaller than forward jumps.

Table 14 DataFlow Jump Backward Analysis - Filter 1

	Back Jumps	Avg. Length	Max Length
Mean	0.61	13.11	16.17
Std Dev	1.09	30.27	40.79
Median	0.00	0.00	0.00
Max	11.00	293.80	567.00

Section 7.3 - Performance Analysis

Measurement Strategy

Baseline configuration

The baseline configuration to compare a series of JavaFlow configurations is defined as a similar hardware system with minimal times between processing nodes. The fundamental structure of the JavaFlow machine is that many processing functions can be deployed across a chip, with the cost of that distribution being the time required to transfer data from one node to another. The baseline machine assumes all the same processing capability with the assumption that each node is adjacent to each other. Obviously this baseline machine could not be physically implemented, but simulation tools can create such an instance. The operation of the simulation is to allow all serial clocks to proceed until there are no more serial messages queued for any nodes. This eliminates the effect of the distance between nodes on the serial network. Mesh messages are then set to transition from one node to another in only one cycle independent of the actual distance in the DataFlow Fabric. This baseline machine performs instructions in dataflow order.

The goal of this baseline definition is the assertion that its performance would approximate that of an optimized Java hardware system or an optimized compiled or JIT (Just in Time) compiled configuration. Due to the complexities of the overall implementation of the Java Virtual Machine and the challenges of technology normalization, proof of this assertion is beyond the scope of this research.

Measurements

The primary performance measurement will be to execute a series of methods described in the following section and measuring the Instructions per Cycle (IPC) by counting the number of instructions executed from the start of a method until a 'Return' instruction is reached, and then dividing by the number of Mesh Cycles. The number of serial clocks between each mesh clock is one of the configuration parameters varied across configurations.

To compare configurations, the IPC for the baseline configuration is normalized to a unity Figure of Merit. Each subsequent measurement is normalized against this baseline to present a Figure of Merit representing a percentage of the baseline IPC.

Configurations

In addition to the Baseline, five system configurations have been proposed and utilized in the performance measurement process. Over 1600 methods are candidates for execution in the simulation environment, and while filters are employed to focus on the most frequently utilized methods, all methods are demonstrated to evaluate the maximum number of instruction paths. Table 15 describes the six configurations used in the measurement processes.

Table 15 Benchmark Configurations

ID	Description
0- Baseline	Collapsed DataFlow machine where dataflow distance is 1 and all serial traffic is moved before next mesh clock.
1 - Compact10	DataFlow mesh 10 units wide, up to 10 serial clocks between each mesh clock .
2 - Compact4	DataFlow mesh 10 units wide; up to 4 serial clocks between each mesh clock
3 - Compact2	DataFlow mesh 10 units wide; up to 2 serial clocks between each mesh clock
4 - Sparse2	DataFlow mesh 10 units wide, up to 2 serial clocks between each mesh clock; each Instruction Node separated by a blank node
5 - Heterogeneous	DataFlow mesh 10 units wide, up to 2 serial clocks between each mesh clock, mesh nodes configured on static instruction mix base (6 arithmetic, 1 floating point, 2 storage, 1 control flow/jump) and automatically assigned

Method Execution

Since the benchmark analysis in Chapter 5 focused on the overall frequency of methods and the individual instructions, the instruction paths were not traced and recorded. Therefore in order to demonstrate the execution of the 1600 methods a pre-established branch prediction methodology was employed to when simulating the method execution. Each method was executed twice with different branch characteristics.

The Branch/Jump predictions applied to the simulation was not complex and used consistently across all 6 configurations. For all forward jumps, the taken/not-taken ratio was 50%. BP1 started with the first forward jump taken while BP2 started with the first jump not taken. In all cases back jumps had a taken percentage of 90%. That means the first 9 executions of a potential jump backwards were taken and then only the 10th execution was not taken or a forward progression through the code.

Filters on methods

Although all 1600 methods were executed in both branch prediction scenarios, the results are filtered by the viability of actually fitting the methods into the DataFlow Fabric.

With a large DataFlow Fabric methods with less than 10 instructions were judged to be not worth the overhead of allocating an Anchor Node and managing their execution. Furthermore methods with greater than 1000 ByteCode instructions would not fit into a DataFlow Fabric unless its size reached 10K nodes.

Therefore three filters were defined and used in the results presentation and are described in Table 16.

Table 16 Filters on Methods

	Selection	# Executions	# Methods
Filter All	All Methods	3210	1605
Filter 1	10< Inst < 1000	1830	915
Filter 2	Top 90% (Dyn) 10 < Inst < 1000	214	107

Simulation Structure

The simulation structure to execute these methods was established in a manner to allow internal configuration parameters to create the 6 configurations described above. While BCEL [54], ASM [55] were used for initial analysis, the basic JAVAP program provided as part of the Java Development environment was used to capture the ByteCode information from the Java ClassFile. This text version of the ByteCode Instruction Stream was then used in the simulation system which both resolved the DataFlow addresses and simulated execution.

The simulation engine shown in Figure 24, shows each ByteCode instruction as an element of an array of 'Instruction Objects.' Both the Serial and Parallel Networks are collapsed into a single static Network object which handles all messages between Instruction Objects. The static Configuration Class is used to maintain configuration metrics and gather results for the measurement processes.

The process is started by a driver which creates a series of Serial Messages for the initial Instruction Object in the array. These serial messages consist of the Tokens described in Chapter 6 with the number or REGISTER_TOKENS equal to the maximum number of registers used by the method. Data from the calling function would then be placed in these registers. These initial serial messages are configured to arrive at the initial Instruction Object sequentially starting with the first serial clock cycle.

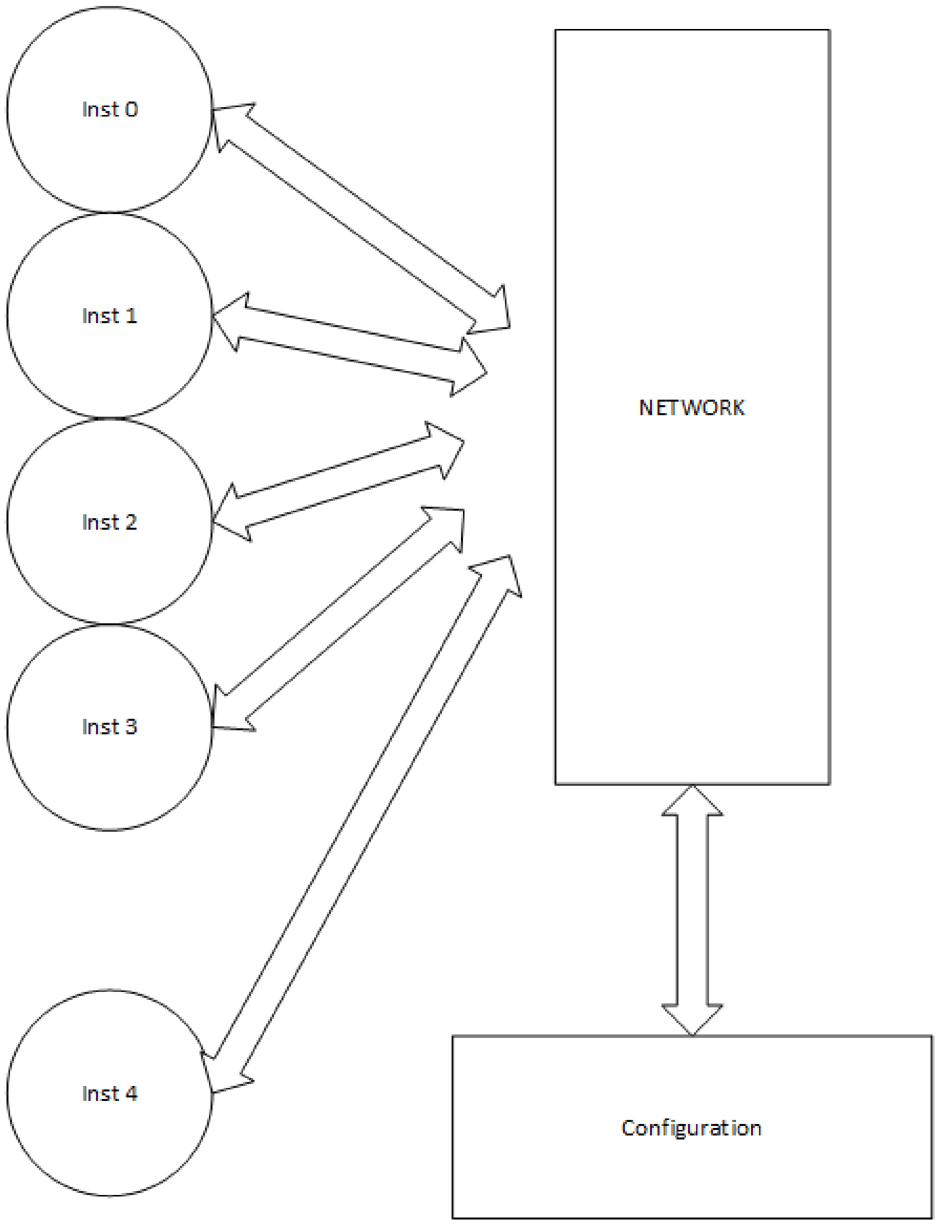


Figure 24 Simulator Class Structure

The Instruction Objects then behave in the manner described in Chapter 6 with the exception of sending messages directly to adjacent nodes or through their own routers, all messages are directed to the Network object who then establishes the processing time and the transit times based on configuration information.

For example, the receipt of a specific serial message may cause an Instruction Node to 'fire.' The result may be the generation of a Mesh Node message to another Instruction Node. This message is sent to the Network where first a lookup is performed to determine the length of processing time necessary for this function. These times are shown in Table 17. The source and destination addresses are used by the Configuration Object to calculate the number of mesh cycles necessary to move the message. These times are then summed to create the total Network time for the message.

The message is then queued inside the Network object and each subsequent mesh or serial cycle causes the appropriate network time to be decremented. Upon reaching zero, the message is then de-queued from the Network and passed along to the appropriate Instruction Object whose 'eMsgToInst' method accepts messages from the network.

This process repeats until a 'Return' instruction is encountered or a timeout occurs. Note that with the branch prediction strategy described above there a few methods which enter an endless loop and do not reach a 'Return' node. To avoid including data from what could be a tight loop, these methods have been filtered from the results.

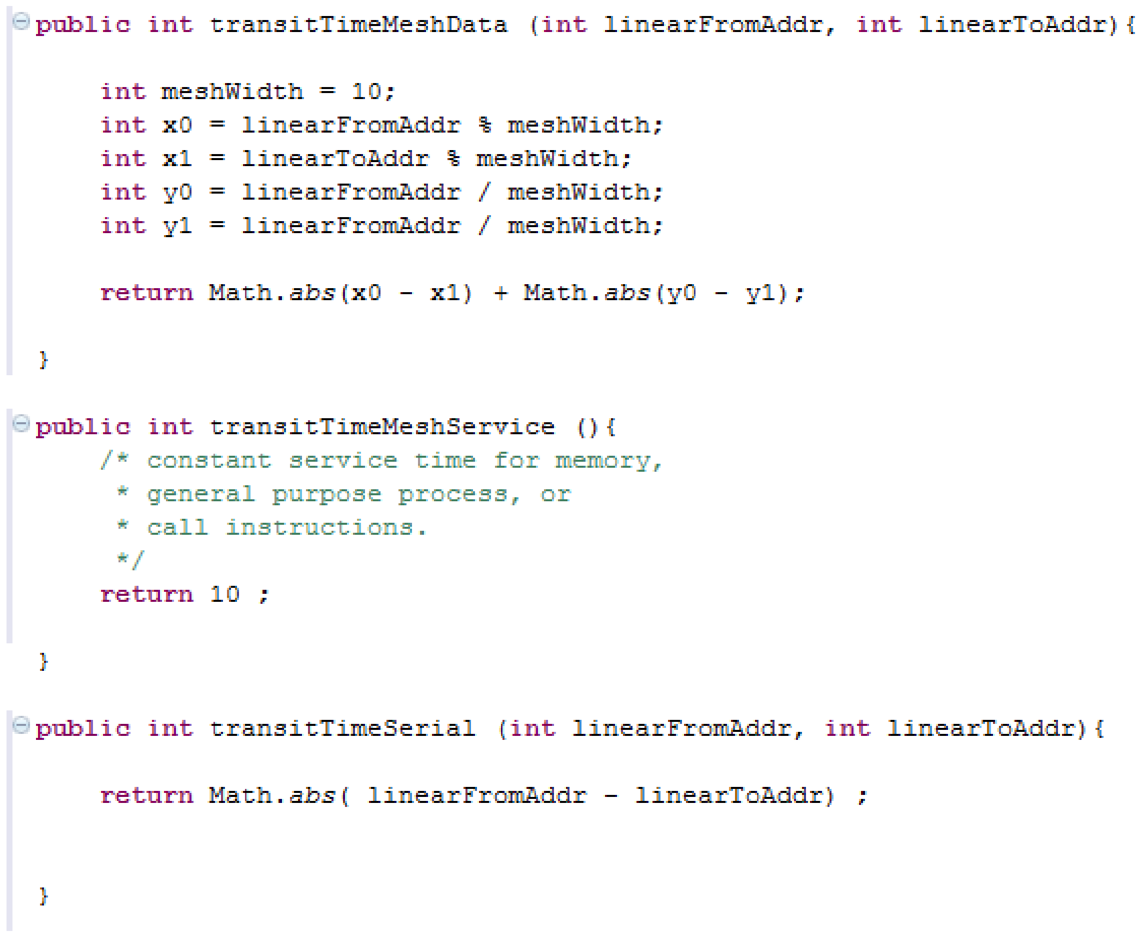
Detailed Assumptions

Each processing node accomplishes its function within a design dependent number of mesh cycles.

Table 17 Execution Cycles per Instruction

Instruction Groups	Mesh Cycles - Execution
Move	1
Floating point arithmetic	10
Integer-Float conversion	5
Special, Logical, Register, Memory	2

Since these assumptions apply to all configurations, the absolute precision of the design detail is not critical. These numbers appear consistent with the 'minimalist' strategy being suggested for the Instruction Nodes. There may be variances in these numbers with word lengths. Some of the more basic register operations may be implementable in a single cycle like the moves. Note that these times do not account of the service times associated with memory, special, or call operations. Note also, that unlike traditional pipelined machines, the instruction decode process has been completed many cycles before any actual processing work is begun.



```java
public int transitTimeMeshData (int linearFromAddr, int linearToAddr){

    int meshWidth = 10;
    int x0 = linearFromAddr % meshWidth;
    int x1 = linearToAddr % meshWidth;
    int y0 = linearFromAddr / meshWidth;
    int y1 = linearFromAddr / meshWidth;

    return Math.abs(x0 - x1) + Math.abs(y0 - y1);

}

public int transitTimeMeshService (){
    /* constant service time for memory,
     * general purpose process, or
     * call instructions.
     */
    return 10 ;

}

public int transitTimeSerial (int linearFromAddr, int linearToAddr){

    return Math.abs( linearFromAddr - linearToAddr) ;


}
```

Figure 25 Network Transit Times

The network transit times account for the time between nodes where the messages are contained in routing circuitry. Figure 25 shows the assumptions used in the performance analysis. Serial messages transition from node to node in one serial clock cycle. The mesh transfer times are dependent on the configuration, and the service times were messages are sent to/from external units are assumed to be constant. The memory times are clearly dependent on the performance of the memory subsystem and may be optimistic. Getting memory data from the inside of the chip to the memory subsystem is a challenge for all designs, and the JavaFlow proposed ring structure provides a high speed transfer from the storage processing nodes to and from the memory subsystem. These are not instruction fetch times as the instructions are already loaded into the Fabric, but rather load and store accesses to the memory subsystem. As described in the previous chapter, normal store operations proceed, while read operations are held up by the service time. The simulation does not highlight any advantages achieved by the separation of the normal heap accesses from the non-ordered method area constant accesses.

The Heterogeneous configuration is perhaps the most interesting in that it does not require each node to process all instructions. This can save significant amounts of hardware although can cause assignment issues with instructions requiring a specific node type having to bypass several nodes in order to match. Furthermore the heterogeneous configuration might have issues as multiple threads are deployed, removed and then reassigned to new methods.

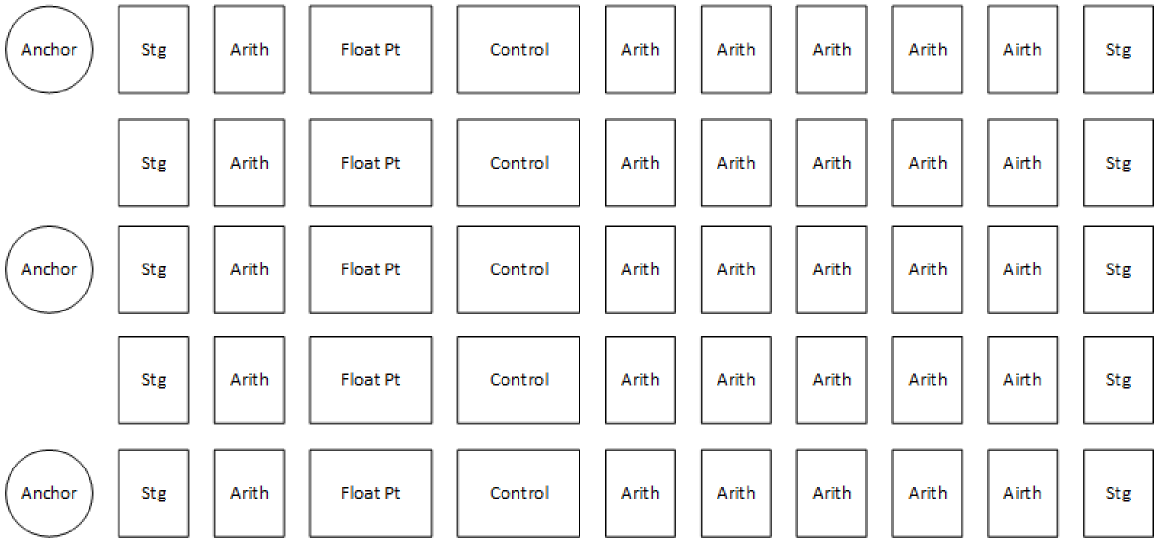


Figure 26 Heterogeneous DataFlow Configuration

Figure 26 shows a configuration of nodes in a 10 wide segment of the DataFlow Fabric where the nodes are assigned by the static instruction mix presented in Chapter 5. The relative width of the nodes indicates the projected amount of circuitry and buffering necessary to implement the function.

As discussed earlier, the Anchor Nodes are set to the side of the DataFlow Fabric and represent the interface point between the General Purpose Processor and the Fabric. By having many Anchors distributed throughout the linear serial network, multiple methods can be loaded without significant interference.

Measurements

Coverage

The first area of measurements is that of coverage. Since the execution is not based on actual trace data, but rather branch predictors, it is necessary to understand the percentage of the instructions which are actually fired in the method to avoid trivial executions. Table 18 shows that for both branch cases, the average coverage is 80% or more of the static instructions in the method.

Table 18 Execution Coverage – All Methods

Inst Exe / Inst Static – 2 Branch Cases		
	BP-1	BP-2
All Cases	83%	80%

Another static measurement is that of the ratio between the number of instructions and the maximum number of DataFlow nodes that are passed in order to house the method. For the Baseline and initial 4 cases this number is fixed due to the homogeneous structure of the DataFlow Fabric. In the case of the Heterogeneous structure, the maximum number of nodes traversed is on average 3.11 times the maximum number of ByteCode Instructions loaded.

Table 19 Ratio of Instructions to Max Node

Case	Inst/MaxNode
Baseline	1.0
Compact10	1.0
Compact4	1.0
Compact2	1.0
Sparse2	2.0
Hetero2	3.11

Table 19 and Table 20 show the details of this Max Node analysis. Table 20 focuses on the Filter 1 case and demonstrates that the mean and median are close and the Standard Deviation is 1.8, although there are some outlying cases were the ratio gets large.

Table 20 Heterogeneous Addressing Detail – Filter 1

Case	Inst/MaxNode
Average	3.11
Median	3.09
Std Dev	1.81
Max	6.53
Min	1.35

Instructions per Cycle and Figure of Merit

Table 21 shows the raw data for the performance runs on the ‘All Method’ case of methods. Because of the different instruction mixes in each method, the median is a better representative IPC number as the Standard Deviation across the IPC’s is very large. Due to the wide variances, Figure of Merits are calculated for each method and then shown in Table 22. Here the normalization to a Figure of Merit of 1 is shown for the Baseline case and the other Figure of Merits are shown. The standard deviations for the Figure of Merits appear more consistent for a single data set.

Note that for the Baseline configuration, some methods have few jumps and many register operations. The configuration of allowing an unlimited number of serial clocks to occur before advancing the Mesh clock, demonstrates some cases where many instructions are able to fire in parallel within a mesh cycle.

Table 21 Raw IPC Data - All Methods

	Raw IPC Data – Filter All				
Case	IPC-Mean	IPC-StdDev	IPC-Median	IPC-Max	IPC-Min
Baseline	0.61	0.84	0.50	19.67	0.13
Compact10	0.54	0.43	0.47	7.26	0.07
Compact4	0.48	0.29	0.43	3.55	0.03
Compact2	0.39	0.19	0.36	1.9	0.02
Sparse2	0.29	.012	0.27	0.99	0.01
Hetero2	0.23	0.11	0.21	1.26	0.01

Table 22 Figure of Merit – Filter All

	Figure of Merit – All Methods		
Case	IPC-Mean	FM	FM StdDev
Baseline	0.61	1.00	
Compact10	0.54	0.96	0.19
Compact4	0.48	0.88	0.19
Compact2	0.39	0.75	0.19
Sparse2	0.29	0.58	0.18
Hetero2	0.23	0.47	0.17

Although not surprising, Table 23 shows that there is minimal correlation between the IPC's in the Heterogeneous configuration and some other measured values such as static and dynamic instruction count, maximum node required, and number of back jumps.

This further supports the point that the variances in the IPC's is dependent on the actual instruction mix and the time utilized to process and transfer the data inside the DataFlow Fabric.

Table 23 Correlations with FM Hetero2 – Filter All

Factor	Correlation
Total I	-0.25
Executed I	-0.21
Max Node	-0.27
Back Jumps	-0.10

Table 24 applies the first filter to the data so that only methods whose size is greater than 10 instructions and less than 1000 are included. Both raw IPC and Figure of Merit data are presented and the conclusion is that there is not much difference between this filter and the data for all methods.

Table 25 shows the same data for Filter 2 where only the methods contributing to 90% of the dynamic processing time are included with the same 10-1000 instruction limits. Again, the IPC for the Baseline improves slightly thus bringing the Figure of Merit for the other cases down a small percentage. Appendix B shows more detailed data for Filter 2 methods.

Table 24 All Data - Filter 1

	Raw IPC Data – Filter-1 (1829 samples) All methods with Static Instructions >10 and <1000			
Case	IPC-Mean	IPC-Median	FM	FM StdDev
Baseline	0.64	0.50	1.00	
Compact10	0.53	0.42	0.86	0.14
Compact4	0.45	0.38	0.77	0.15
Compact2	0.37	0.32	0.66	0.16
Sparse2	0.27	0.25	0.50	0.16
Hetero2	0.23	0.22	0.44	0.15

Table 25 All Data - Filter 2

	Raw IPC Data – Filter-2 (214 samples) Top 90% Methods with Static Instructions >10 and <1000			
Case	IPC-Mean	IPC-Median	FM	FM StdDev
Baseline	0.72	0.48	1.00	
Compact10	0.53	0.39	0.82	0.14
Compact4	0.44	0.35	0.74	0.17
Compact2	0.36	0.30	0.63	0.18
Sparse2	0.26	0.24	0.49	0.17
Hetero2	0.23	0.22	0.43	0.17

Parallelism

Traditional DataFlow machines were designed to achieve parallelism in the execution of instructions. Although this was not a primary design objective of JavaFlow, the basic level of parallelism was assessed for each method execution. For each Mesh Cycle during the execution of a method counters were kept to identify if there was a single instruction executing or if there were 2 or more instructions executing. Only execution times were used with service times not included in this analysis. Table 26 shows the average percentage of mesh clock cycles where 2 or more Instruction Nodes were performing execution.

Table 26 Parallelism - All Methods

Case	Average % Mesh Cycles with 2 or more Instructions Executing
Baseline	40%
Compact10	37%
Compact4	33%
Compact2	24%
Sparse2	13%
Hetero2	12%

Measurements vs Top 4 Spec Benchmark Methods

With the measurements of IPC and resulting Figure of Merit, it is now possible to go back to the data presented in Chapter 5 to project how the JavaFlow machine would handle the SPEC benchmarks. Table 27 and Table 28 show are comparable to Table 3 and

Table 4 from Chapter 5. Some benchmarks were eliminated due to size or timeouts in the simulation. The identified methods comprise 31% to 100% of the SpecJvm2008 ByteCode executions and 17% to 84% of the SpecJvm98 ByteCode executions.

The columns in Table 27 and Table 28 are:

- Benchmark, method
- Total I - Total static instruction count for the method
- Sparser N - Ratio of instruction count to maximum node required in Heterogeneous configuration
- fmM The Figure of Merit for each configuration N

The total number of instructions and total nodes spanned in the Heterogeneous case are totaled. The Figures of Merit as described above are displayed and averaged. The conclusion is that these critical methods can be resident in a 10,000 Instruction Node fabric, and when considering individual benchmarks, a significantly smaller DataFlow Fabric could be effective in executing the methods.

The average Figure of Merit on the Heterogeneous is slightly smaller than the averages shown above, but still above 35%. The reasons for the decrease is that the baseline IPC for the two cases are 0.60 and 0.52 vs the 0.72 shown in Table 25

Table 27 Figure of Merit on Top 4 SpecJvm2008 Benchmarks

	Total I	Sparser N	fm0	fm1	fm2	fm3	fm4	fm5
SpecJvm2008								
compress								
compress()V	165	398	100%	78%	66%	52%	34%	30%
output(I)V	208	448	100%	67%	54%	40%	27%	25%
decompress()V	119	338	100%	76%	71%	61%	44%	37%
update([B)V	10	18	100%	92%	88%	85%	73%	65%
crypto.signverify								
mul([I[II[II)V	72	148	100%	69%	60%	50%	34%	33%
submul_1([II[III)I	73	148	100%	67%	61%	53%	39%	39%
sha(IIIII[BI)[I	315	578	100%	65%	54%	42%	28%	30%
sha(IIIIIIII[BI)[I	288	518	100%	55%	45%	33%	21%	23%
mpegaudio								
dequantize_sample([[FI	551	1358	100%	74%	59%	44%	29%	25%
huffman_decode(II)V	399	1058	100%	81%	77%	67%	51%	45%
hybrid(II)V	580	1288	100%	54%	32%	19%	11%	10%
scumark.fft.large								
bitreverse([D)V	86	168	100%	77%	56%	39%	23%	23%
transform_internal([DI)'	251	608	100%	90%	69%	50%	47%	39%
nextDouble()D	71	178	100%	83%	78%	71%	56%	47%
inverse([D)V	31	68	100%	79%	71%	61%	45%	43%
scimark.lu.large								
factor([[D[I)I	162	358	100%	64%	46%	31%	19%	18%
nextDouble()D	71	178	100%	83%	78%	71%	56%	47%
scimark.monte_carlo								
submul_1([II[III)I	73	148	100%	67%	61%	53%	39%	39%
sha(IIIII[BI)[I	315	578	100%	65%	54%	42%	28%	30%
integrate(I)D	39	118	100%	72%	64%	55%	43%	40%
nextDouble()D	71	178	100%	83%	78%	71%	56%	47%
scimark.sor.large								
submul_1([II[III)I	73	148	100%	67%	61%	53%	39%	39%
nextDouble()D	71	178	100%	83%	78%	71%	56%	47%
execute(D[[DI)D	111	258	100%	36%	18%	10%	6%	5%
scimark.sparse.large								
nextDouble()D	71	178	100%	83%	78%	71%	56%	47%
Sum/Mean	4276	9640		72%	62%	52%	38%	35%

Table 28 Figure of Merit on Top 4 SpecJvm98 Benchmarks

	Total I	Sparser N	fm0	fm1	fm2	fm3	fm4	fm5
SpecJvm98								
_201_compress								
compress()V	178	398	100%	78%	69%	58%	41%	39%
output(I)V	216	438	100%	69%	55%	40%	25%	26%
decompress()V	181	408	100%	100%	87%	68%	52%	31%
getbyte()I	24	58	100%	100%	86%	66%	45%	38%
_202_jess								
runTestsVaryRight(Lspe	51	128	100%	81%	73%	62%	47%	46%
data_equals(Lspec/benc	24	68	100%	76%	70%	59%	50%	40%
equals(Lspec/benchmar	46	158	100%	117%	100%	70%	54%	35%
equals(Ljava/lang/Objec	37	118	100%	117%	100%	70%	54%	35%
_209_db								
compareTo(Ljava/lang/S	77	178	100%	74%	64%	52%	36%	34%
shell_sort(I)V	88	218	100%	77%	67%	55%	38%	34%
elementAt(I)Ljava/lang/	23	108						
_222_mpegaudio								
Ä£(Lspec/benchmarks/_	100	208	100%	90%	80%	69%	46%	46%
read([BII)I	66	158	100%	71%	58%	43%	28%	26%
l([SI)I	346	888	100%	58%	45%	32%	20%	17%
_227_mtrt								
FindTreeNode(Lspec/be	97	408	100%	86%	84%	79%	64%	57%
Intersect(Lspec/benchm	701	2608	100%	71%	60%	46%	32%	23%
Combine(Lspec/benchm	35	158	100%	77%	75%	70%	61%	65%
_228_jack								
<init>([C)V	17	38	100%	84%	79%	68%	56%	49%
nextElement()Ljava/util/	43	118	100%	74%	67%	57%	44%	36%
Move(CLjava/util/Vecto	156	468	100%	70%	55%	40%	26%	20%
getNextTokenFromStre	360	1038	100%	73%	66%	51%	44%	38%
Sum/Mean	2866	8368		82%	72%	58%	43%	37%

Chapter 8: Conclusions

Although JavaFlow is not a complete machine implementation, the concepts established and the results obtained suggest that it has met the objectives set forth and should be seriously considered as an approach for future system implementations.

Attaining 40% of the baseline performance target with a set of heterogeneous cores in the DataFlow fabric with a 3.1 ratio of instructions to nodes is demonstration of success. In addition to achieving acceptable performance in general, the results of the performance measurements are tied back to the Benchmark analysis and show that key methods can be made resident in the DataFlow Fabric and similar performance can be achieved.

With the ability to load multiple methods into the DataFlow Fabric at the same time, these methods can be executing simultaneously. Since the network traffic is localized to the area of the Fabric of each method it is reasonable to use an argument of superposition to claim that the overall Instructions per Cycle for the system would be the sum of the individual Instructions per Cycle for each method.

The demonstrated capability to manage the loading and address resolution of ByteCode instructions without centralized control represents a novel approach for future tiled architecture systems like JavaFlow.

The performance, parallelism, and instruction execution unit utilization parameters measured in Chapter 7 show attractive performance, which then allows the focus on the non-quantified objectives of reducing design complexity, power constraints, on-chip wiring issues, data locality, and management of a large number of Instruction Nodes.

Finally enhancement items described along with continued tuning of the DataFlow Fabric configuration should allow further performance gains.

The evaluation of the non-quantifiable aspects of this system requires a new set of metrics. Replicating a minimalist design of a core thousands of times across a chip saves development expense and complexity. Leaving many of these cores unused and unpowered (e.g., dark silicon) while executing a program represents power savings that would not otherwise occur. The globally asynchronous, locally synchronous design removes the requirement for precise clock control across a chip and can minimize skew when transferring over short distances.

Appendices

Appendix A - ByteCode Instructions

Appendix A provides a series of tables listing all ByteCode instructions architected in the Java Virtual Machine [4]. Each table shows a group of instructions whose behavior is similar. The instruction groups are defined in the captions of each table.

Each table has the following columns:

- OpCode: The abbreviation of the operation code of the ByteCode instruction
- Stack Before > After: The contents of the Java Stack before and after the execution of the instruction. The 'Before' values are consumed by the instructions and the 'After' values represent the results of the computation.
- Pop: The number of variables that are 'popped' or consumed by the instruction as inputs. In the JavaFlow machine, these values are provided by producer nodes elsewhere in the DataFlow Fabric
- Push: The number of variables 'pushed' or produced by the instruction as a result of the computation. In the JavaFlow machine these values are sent to other nodes in the DataFlow fabric.
- Description: A brief description of computation implemented by each instruction.

Table 29 ByteCode Floating Point Conversion Instructions

	OpCode	Stack Before > After	pop	push	Description
Floating Pt Conversion	d2f	value → result	1	1	convert a double to a float
	d2i	value → result	1	1	convert a double to an int
	d2l	value → result	1	1	convert a double to a long
	f2d	value → result	1	1	convert a float to a double
	f2i	value → result	1	1	convert a float to an int
	f2l	value → result	1	1	convert a float to a long
	l2d	value → result	1	1	convert a long to a double
	l2f	value → result	1	1	convert a long to a float
	i2b	value → result	1	1	convert an int into a byte
	i2c	value → result	1	1	convert an int into a character
	i2d	value → result	1	1	convert an int into a double
	i2f	value → result	1	1	convert an int into a float
	i2l	value → result	1	1	convert an int into a long
	i2s	value → result	1	1	convert an int into a short
	l2i	value → result	1	1	convert a long to a int

Table 30 ByteCode Arithmetic/Integer Instructions

	OpCode	Stack Before > After	pop	push	Description
Arithmetic/Integer	iadd	value1, value2 → result	2	1	add two ints
	iand	value1, value2 → result	2	1	perform a bitwise and on two integers
	idiv	value1, value2 → result	2	1	divide two integers
	imul	value1, value2 → result	2	1	multiply two integers
	ineg	value → result	1	1	negate int
	ior	value1, value2 → result	2	1	bitwise int or
	irem	value1, value2 → result	2	1	logical int remainder
	ishl	value1, value2 → result	2	1	int shift left
	ishr	value1, value2 → result	2	1	int arithmetic shift right
	isub	value1, value2 → result	2	1	int subtract
	iushr	value1, value2 → result	2	1	int logical shift right
	ixor	value1, value2 → result	2	1	int xor
	ladd	value1, value2 → result	2	1	add two longs
	land	value1, value2 → result	2	1	bitwise and of two longs
	lmul	value1, value2 → result	2	1	multiply two longs
	lneg	value → result	1	1	negate a long
	lor	value1, value2 → result	2	1	bitwise or of two longs
	lrem	value1, value2 → result	2	1	remainder of division of two longs
	lshl	value1, value2 → result	2	1	bitwise shift left of a long *value1* by int *value2* positions
	lshr	value1, value2 → result	2	1	bitwise shift right of a long *value1* by int *value2* positions
	lsub	value1, value2 → result	2	1	subtract two longs
	lushr	value1, value2 → result	2	1	bitwise shift right of a long *value1* by int *value2* positions, unsigned
	lxor	value1, value2 → result	2	1	bitwise exclusive or of two longs

Table 31 ByteCode Arithmetic/Move Instructions

	OpCode	Stack Before > After	pop	push	Description
Arithmetic/Move	aconst_null	→ null	0	1	push a *null* reference onto the stack
	bipush	→ value	0	1	push a *byte* onto the stack as an integer *value*
	dconst_0	→ 0.0	0	1	push the constant *0.0* onto the stack
	dconst_1	→ 1.0	0	1	push the constant *1.0* onto the stack
	dup	value → value, value	1	2	duplicate the value on top of the stack
	dup_x1	value2, value1 → value1, value2, value1	3	5	insert a copy of the top value into the stack two values from the top
	dup_x2	value3, value2, value1 → value1, value3, value2, value1	4	6	insert a copy of the top value into the stack two or three values from the top
	dup2	{value2, value1} → {value2, value1}, {value2, value1}	2	3	duplicate top two stack words
	dup2_x1	value3, {value2, value1} → {value2, value1}, value3, {value2, value1}	3	4	duplicate two words and insert beneath third word
	dup2_x2	{value4, value3}, {value2, value1} → {value2, value1}, {value4, value3}, {value2, value1}	2	4	duplicate two words and insert beneath fourth word
	fconst_0	→ 0.0f	0	1	push *0.0f* on the stack
	fconst_1	→ 1.0f	0	1	push *1.0f* on the stack
	fconst_2	→ 2.0f	0	1	push *2.0f* on the stack
	iconst_0	→ 0	0	1	load the int value 0 onto the stack
	iconst_1	→ 1	0	1	load the int value 1 onto the stack
	iconst_2	→ 2	0	1	load the int value 2 onto the stack
	iconst_3	→ 3	0	1	load the int value 3 onto the stack
	iconst_4	→ 4	0	1	load the int value 4 onto the stack
	iconst_5	→ 5	0	1	load the int value 5 onto the stack
	iconst_m1	→ -1	0	1	load the int value -1 onto the stack
	lconst_0	→ 0L	0	1	push the long 0 onto the stack
	lconst_1	→ 1L	0	1	push the long 1 onto the stack
	pop	value →	1	0	discard the top value on the stack
	pop2	{value2, value1} →	1	0	discard the top two values on the stack
	sipush	→ value	0	1	push a short onto the stack
	swap	value2, value1 → value1, value2	2	2	swaps two top words on the stack

Table 32 ByteCode Floating Point Arithmetic Instructions

	OpCode	Stack Before > After	pop	push	Description
Floating Pt Arithmetic	dadd	value1, value2 → result	2	1	add two doubles
	dcmpg	value1, value2 → result	2	1	compare two doubles
	dcmpl	value1, value2 → result	2	1	compare two doubles
	ddiv	value1, value2 → result	2	1	divide two doubles
	dmul	value1, value2 → result	2	1	multiply two doubles
	dneg	value → result	1	1	negate a double
	drem	value1, value2 → result	2	1	get the remainder from a division between two doubles
	dsub	value1, value2 → result	2	1	subtract a double from another
	fadd	value1, value2 → result	2	1	add two floats
	fcmpg	value1, value2 → result	2	1	compare two floats
	fcmpl	value1, value2 → result	2	1	compare two floats
	fdiv	value1, value2 → result	2	1	divide two floats
	fmul	value1, value2 → result	2	1	multiply two floats
	fneg	value → result	1	1	negate a float
	frem	value1, value2 → result	2	1	get the remainder from a division between two floats
	fsub	value1, value2 → result	2	1	subtract two floats
	lcmp	value1, value2 → result	2	1	compare two longs values
	ldiv	value1, value2 → result	2	1	divide two longs

Table 33 ByteCode Control Flow Instructions

	OpCode	Stack Before > After	pop	push	Description
Control Flow Change	goto	[no change]	0	0	goes to another instruction at *branchoffset*
	goto_w	[no change]	0	0	goes to another instruction at *branchoffset*
	if_acmpeq	value1, value2 →	2	0	if references are equal, branch to instruction at *branchoffset*
	if_acmpne	value1, value2 →	2	0	if references are not equal, branch to instruction at*branchoffset*
	if_icmpeq	value1, value2 →	2	0	if ints are equal, branch to instruction at *branchoffset*
	if_icmpge	value1, value2 →	2	0	if *value1* is greater than or equal to *value2*, branch to instruction at *branchoffset*
	if_icmpgt	value1, value2 →	2	0	if *value1* is greater than *value2*, branch to instruction at*branchoffset*
	if_icmple	value1, value2 →	2	0	if *value1* is less than or equal to *value2*, branch to instruction at *branchoffset*
	if_icmplt	value1, value2 →	2	0	if *value1* is less than *value2*, branch to instruction at*branchoffset*
	if_icmpne	value1, value2 →	2	0	if ints are not equal, branch to instruction at *branchoffset*
	ifeq	value →	1	0	if *value* is 0, branch to instruction at *branchoffset*
	ifge	value →	1	0	if *value* is greater than or equal to 0, branch to instruction at*branchoffset*
	ifgt	value →	1	0	if *value* is greater than 0, branch to instruction at*branchoffset*
	ifle	value →	1	0	if *value* is less than or equal to 0, branch to instruction at*branchoffset*
	iflt	value →	1	0	if *value* is less than 0, branch to instruction at *branchoffset*
	ifne	value →	1	0	if *value* is not 0, branch to instruction at *branchoffset*
	ifnonnull	value →	1	0	if *value* is not null, branch to instruction at *branchoffset*
	ifnull	value →	1	0	if *value* is null, branch to instruction at *branchoffset*

Table 34 ByteCode Call Instructions

	OpCode	Stack Before > After	pop	push	Description
Call	invokeinterface	objectref, [arg1, arg2, ...] →	n	0	invokes an interface method on object *objectref*
	invokespecial	objectref, [arg1, arg2, ...] →	n	1	invoke instance method on object *objectref*
	invokestatic	[arg1, arg2, ...] →	n	1	invoke a static method
	invokevirtual	objectref, [arg1, arg2, ...] →	n	1	invoke virtual method on object *objectref*

Table 35 ByteCode Return Instructions

	OpCode	Stack Before > After	pop	push	Description
Return	areturn	objectref → [empty]	1	0	return a reference from a method
	athrow	objectref → [empty], objectref	1	0	throws an error or exception
	dreturn	value → [empty]	1	0	return a double from a method
	freturn	value → [empty]	1	0	return a float
	ireturn	value → [empty]	1	0	return an integer from a method
	lreturn	value → [empty]	1	0	return a long value
	return	→ [empty]	0	0	return void from method

Table 36 ByteCode Memory Constant Instructions

	OpCode	Stack Before > After	pop	push	Description
Memory Constant	ldc	→ value	0	1	push a constant *#index* from a constant pool onto the stack
	ldc_w	→ value	0	1	push a constant *#index* from a constant pool
	ldc2_w	→ value	0	1	push a constant *#index* from a constant pool

Table 37 ByteCode Memory Read Instructions

	OpCode	Stack Before > After	pop	push	Description
Memory Read	aaload	arrayref, index → value	2	1	load onto the stack a reference from an array
	baload	arrayref, index → value	2	1	load a byte or Boolean value from an array
	caload	arrayref, index → value	2	1	load a char from an array
	daload	arrayref, index → value	2	1	load a double from an array
	faload	arrayref, index → value	2	1	load a float from an array
	getfield	objectref → value	1	1	get a field *value* of an object *objectref*, where the field is identified by field reference in the constant pool *index* (*index1* << *8* + *index2*)
	getstatic	→ value	0	1	get a static field *value* of a class, where the field is identified by field reference in the constant pool *index* (*index1* << *8* + *index2*)
	iaload	arrayref, index → value	2	1	load an int from an array
	laload	arrayref, index → value	2	1	load a long from an array
	saload	arrayref, index → value	2	1	load short from array

Table 38 ByteCode Memory Write Instructions

	OpCode	Stack Before > After	pop	push	Description
Memory Write	aastore	arrayref, index, value →	3	0	store into a reference in an array
	bastore	arrayref, index, value →	3	0	store a byte or Boolean value into an array
	castore	arrayref, index, value →	3	0	store a char into an array
	dastore	arrayref, index, value →	3	0	store a double into an array
	fastore	arrayref, index, value →	3	0	store a float in an array
	iastore	arrayref, index, value →	3	0	store an int into an array
	lastore	arrayref, index, value →	3	0	store a long to an array
	putfield	objectref, value →	2	0	set field to *value* in an object *objectref*, where the field is identified by a field reference *index* in constant pool (*indexbyte1* << *8* + *indexbyte2*)
	putstatic	value →	1	0	set static field to *value* in a class, where the field is identified by a field reference *index* in constant pool (*indexbyte1* << *8* + *indexbyte2*)
	sastore	arrayref, index, value →	3	0	store short to array

Table 39 ByteCode Local Read Instructions

	OpCode	Stack Before > After	pop	push	Description
Local Increment	iinc	[No change]	0	0	increment local variable *#index* by signed byte *const*
Local Read	aload	→ objectref	0	1	load a reference onto the stack from a local variable *#index*
	aload_0	→ objectref	0	1	load a reference onto the stack from local variable 0
	aload_1	→ objectref	0	1	load a reference onto the stack from local variable 1
	aload_2	→ objectref	0	1	load a reference onto the stack from local variable 2
	aload_3	→ objectref	0	1	load a reference onto the stack from local variable 3
	dload	→ value	0	1	load a double *value* from a local variable *#index*
	dload_0	→ value	0	1	load a double from local variable 0
	dload_1	→ value	0	1	load a double from local variable 1
	dload_2	→ value	0	1	load a double from local variable 2
	dload_3	→ value	0	1	load a double from local variable 3
	fload	→ value	0	1	load a float *value* from a local variable *#index*
	fload_0	→ value	0	1	load a float *value* from local variable 0
	fload_1	→ value	0	1	load a float *value* from local variable 1
	fload_2	→ value	0	1	load a float *value* from local variable 2
	fload_3	→ value	0	1	load a float *value* from local variable 3
	iload	→ value	0	1	load an int *value* from a local variable *#index*
	iload_0	→ value	0	1	load an int *value* from local variable 0
	iload_1	→ value	0	1	load an int *value* from local variable 1
	iload_2	→ value	0	1	load an int *value* from local variable 2
	iload_3	→ value	0	1	load an int *value* from local variable 3
	lload	→ value	0	1	load a long value from a local variable *#index*
	lload_0	→ value	0	1	load a long value from a local variable 0
	lload_1	→ value	0	1	load a long value from a local variable 1
	lload_2	→ value	0	1	load a long value from a local variable 2
	lload_3	→ value	0	1	load a long value from a local variable 3

Table 40 ByteCode Local Write Instructions

	OpCode	Stack Before > After	pop	push	Description
Local Write	astore	objectref →	1	0	store a reference into a local variable *#index*
	astore_0	objectref →	1	0	store a reference into local variable 0
	astore_1	objectref →	1	0	store a reference into local variable 1
	astore_2	objectref →	1	0	store a reference into local variable 2
	astore_3	objectref →	1	0	store a reference into local variable 3
	dstore	value →	1	0	store a double *value* into a local variable *#index*
	dstore_0	value →	1	0	store a double into local variable 0
	dstore_1	value →	1	0	store a double into local variable 1
	dstore_2	value →	1	0	store a double into local variable 2
	dstore_3	value →	1	0	store a double into local variable 3
	fstore	value →	1	0	store a float *value* into a local variable *#index*
	fstore_0	value →	1	0	store a float *value* into local variable 0
	fstore_1	value →	1	0	store a float *value* into local variable 1
	fstore_2	value →	1	0	store a float *value* into local variable 2
	fstore_3	value →	1	0	store a float *value* into local variable 3
	istore	value →	1	0	store int *value* into variable *#index*
	istore_0	value →	1	0	store int *value* into variable 0
	istore_1	value →	1	0	store int *value* into variable 1
	istore_2	value →	1	0	store int *value* into variable 2
	istore_3	value →	1	0	store int *value* into variable 3
	lstore	value →	1	0	store a long *value* in a local variable *#index*
	lstore_0	value →	1	0	store a long *value* in a local variable 0
	lstore_1	value →	1	0	store a long *value* in a local variable 1
	lstore_2	value →	1	0	store a long *value* in a local variable 2
	lstore_3	value →	1	0	store a long *value* in a local variable 3

Table 41 ByteCode Special Instructions

	OpCode	Stack Before > After	pop	push	Description
Special	anewarray	count → arrayref	1	1	create a new array of references
	arraylength	arrayref → length	1	1	get the length of an array
	checkcast	objectref → objectref	1	1	checks whether an *objectref* is of a certain type
	instanceof	objectref → result	1	1	determines if an object *objectref* is of a given type
	jsr	→ address	0	1	jump to subroutine at *branchoffset*
	jsr_w	→ address	0	1	jump to subroutine at *branchoffset*
	lookupswitch	key →	1	0	a target address is looked up from a table using a key and execution continues from the instruction at that address
	monitorenter	objectref →	1	0	enter monitor for object
	monitorexit	objectref →	1	0	exit monitor for object
	multianewarray	count1, [count2,...] → arrayref	1	1	create a new array
	new	→ objectref	0	1	create new object
	newarray	count → arrayref	1	1	create new array
	nop	[No change]	0	0	perform no operation
	ret	[No change]	0	0	continue execution from address taken from a local variable#*index*
	tableswitch	index →	1	0	continue execution from an address in the table at offset*index*
	wide	[same as for corresponding instructions]			execute *opcode* , where *opcode* is either iload, fload, aload, lload, dload, istore, fstore, astore, lstore, dstore, or ret, but assume the *index* is 16 bit; or execute iinc, where the *index* is 16 bits and the constant to increment by is a signed 16 bit short

APPENDIX B - DATA FROM TOP 90% METHODS

The following Tables show the raw data from the simulations of the methods contributing to the top 90% of the operations in the SPEC benchmarks.

The columns in these tables represent:

- Name of the method. (Note that to save space, the name of the class to which the method belongs is eliminated)
- Total number of static instructions
- Total number of nodes in the Heterogeneous configuration
- Ratio of nodes to static instructions
- Number of instruction executed in Branch case 0
- Mesh cycles required;
- IPC (Instructions per cycle) for the 6 configurations (Note that the Figure of Merit reported in Chapter 7 is the ratio of each of these values to the value in IPC-0.

Table 42 Top 90% methods - part 1

Method	Total Inst	Hetero Nodes	N/I	Inst Exec	Mesh Cycles	IPC-0	IPC-1	IPC-2	IPC-3	IPC-4	IPC-5
divide([II[II)V	134	248	1.85	223	293	0.76	0.31	0.21	0.13	0.08	0.08
submul_1([II[III)I	73	148	2.03	611	895	0.68	0.46	0.41	0.36	0.26	0.27
sha(IIIII[BI)[I	315	578	1.83	2365	1980	1.19	0.77	0.65	0.50	0.34	0.36
sha(IIIIIIII[BI)[I	288	518	1.80	1993	1116	1.79	0.98	0.80	0.59	0.37	0.41
equals(Ljava/lang/Object;)Z	15	58	3.87	11	38	0.29	0.29	0.28	0.26	0.21	0.17
<init>()V	13	28	2.15	13	20	0.65	0.48	0.46	0.42	0.36	0.35
charAt(I)C	19	58	3.05	14	52	0.27	0.26	0.25	0.24	0.17	0.15
compareTo(Ljava/lang/String;)I	77	178	2.31	39	58	0.67	0.50	0.43	0.35	0.24	0.23
equals(Ljava/lang/Object;)Z	48	118	2.46	5	7	0.71	0.83	0.62	0.38	0.33	0.23
getBytes(II[BI)V	55	158	2.87	7	20	0.35	0.33	0.30	0.26	0.20	0.16
getChars([CI)V	10	28	2.80	10	20	0.50	0.43	0.42	0.38	0.34	0.28
hashCode()I	38	78	2.05	10	39	0.26	0.24	0.22	0.18	0.13	0.11
toLowerCase(Ljava/util/Locale;)Ljava/lang/String;	246	668	2.72	6	20	0.30	0.30	0.25	0.21	0.18	0.13
<init>(IIII)V	60	248	4.13	22	46	0.48	0.41	0.38	0.33	0.28	0.25
<init>(IIII[BI)V	24	58	2.42	23	26	0.88	0.85	0.74	0.59	0.43	0.38
wrap([BII)Ljava/nio/ByteBuffer;	12	48	4.00	7	17	0.41	0.37	0.35	0.32	0.32	0.24
<init>([BIIIIIZ)V	13	28	2.15	13	15	0.87	0.59	0.54	0.48	0.48	0.42
get()B	23	58	2.52	9	34	0.26	0.26	0.26	0.25	0.21	0.18
put([BII)Ljava/nio/ByteBuffer;	35	88	2.51	11	37	0.30	0.25	0.25	0.24	0.21	0.19
get()C	23	58	2.52	9	34	0.26	0.26	0.26	0.25	0.21	0.18
checkName(Ljava/lang/String;)V	81	358	4.42	62	96	0.65	0.47	0.41	0.36	0.26	0.22
<init>(Ljava/nio/charset/Charset;FF[B)V	61	178	2.92	26	34	0.76	0.59	0.54	0.46	0.37	0.34
encode(Ljava/nio/CharBuffer;Ljava/nio/ByteBuffer;Z)Ljav	132	518	3.92	22	37	0.59	0.49	0.40	0.31	0.23	0.18
equals(Ljava/lang/Object;)Z	67	298	4.45	5	7	0.71	0.83	0.62	0.42	0.33	0.23
fill([JJ)V	15	38	2.53	18	30	0.60	0.50	0.42	0.35	0.27	0.23
get(Ljava/lang/Object;)Ljava/lang/Object;	43	148	3.44	5	10	0.50	0.56	0.50	0.36	0.31	0.19
<init>(Ljava/util/Hashtable;)V	15	38	2.53	15	35	0.43	0.43	0.41	0.37	0.30	0.26

Table 43 Top 90% methods - part 2

Method	Total Inst	Hetero Nodes	N/I	Inst Exec	Mesh Cycles	IPC-0	IPC-1	IPC-2	IPC-3	IPC-4	IPC-5
nextElement()Ljava/util/Map$Entry;	43	118	2.74	8	33	0.24	0.24	0.24	0.22	0.19	0.16
<init>(Ljava/util/Hashtable;)V	12	28	2.33	12	21	0.57	0.46	0.44	0.41	0.35	0.32
nextElement()Ljava/lang/Object;	14	38	2.71	14	55	0.25	0.23	0.22	0.21	0.19	0.17
addElement(Ljava/lang/Object;)V	24	38	1.58	24	46	0.52	0.40	0.39	0.36	0.27	0.35
elementAt(I)Ljava/lang/Object;	23	108	4.70	19	71	0.27	0.24	0.24	0.22	0.18	0.17
removeAllElements()V	23	48	2.09	27	58	0.47	0.34	0.32	0.28	0.23	0.23
nextElement()Ljava/lang/Object;	37	98	2.65	24	67	0.36	0.31	0.30	0.28	0.23	0.21
hget1bit()I	22	48	2.18	22	34	0.65	0.54	0.50	0.45	0.37	0.39
huffman_decoder(Ljavazoom/jl/decoder/huffcodetab;[I[	305	738	2.42	15	19	0.79	0.79	0.65	0.47	0.39	0.31
dequantize_sample([[FII)V	551	1358	2.46	222	685	0.32	0.24	0.19	0.14	0.09	0.08
huffman_decode(II)V	399	1058	2.65	202	534	0.38	0.31	0.29	0.25	0.19	0.17
reorder([[FII)V	239	518	2.17	48	154	0.31	0.24	0.20	0.15	0.10	0.09
appendSamples(I[F)V	50	108	2.16	50	136	0.37	0.31	0.29	0.25	0.18	0.18
compute_pcm_samples0(Ljavazoom/jl/decoder/Obuffer	188	538	2.86	191	254	0.75	0.68	0.56	0.45	0.29	0.22
compute_pcm_samples1(Ljavazoom/jl/decoder/Obuffer	188	538	2.86	191	254	0.75	0.68	0.56	0.45	0.29	0.22
compute_pcm_samples15(Ljavazoom/jl/decoder/Obuffe	188	538	2.86	191	254	0.75	0.68	0.56	0.45	0.29	0.22
compute_pcm_samples3(Ljavazoom/jl/decoder/Obuffer	190	538	2.83	193	254	0.76	0.68	0.56	0.44	0.29	0.22
compute_pcm_samples5(Ljavazoom/jl/decoder/Obuffer	188	538	2.86	191	254	0.75	0.68	0.56	0.45	0.29	0.22
input_samples([F)V	19	58	3.05	21	50	0.42	0.40	0.34	0.29	0.20	0.16
output(I)V	216	438	2.03	88	116	0.76	0.51	0.44	0.36	0.27	0.26
decompress()V	181	408	2.25	9	13	0.69	0.69	0.60	0.47	0.36	0.21
getcode()I	172	328	1.91	147	242	0.61	0.54	0.51	0.45	0.31	0.34
getbyte()I	24	58	2.42	22	53	0.42	0.31	0.31	0.29	0.29	0.26
putbyte(B)V	12	18	1.50	12	25	0.48	0.48	0.46	0.43	0.29	0.31
Call(Lspec/benchmarks/_202_jess/jess/ValueVector;Lspe	23	78	3.39	123	212	0.58	0.39	0.35	0.29	0.23	0.17
clone()Ljava/lang/Object;	25	58	2.32	25	52	0.48	0.45	0.42	0.40	0.33	0.30
Execute(Lspec/benchmarks/_202_jess/jess/ValueVector	140	538	3.84	20	49	0.41	0.29	0.22	0.16	0.10	0.06

Table 44 Top 90% methods - part 3

Method	Total Inst	Hetero Nodes	N/I	Inst Exec	Mesh Cycles	IPC-0	IPC-1	IPC-2	IPC-3	IPC-4	IPC-5
SimpleExecute(Lspec/benchmarks/_202_jess/jess/Value	49	168	3.43	19	37	0.51	0.43	0.40	0.40	0.30	0.25
Eval(Lspec/benchmarks/_202_jess/jess/Value;Lspec/ber	59	228	3.86	284	475	0.60	0.43	0.35	0.27	0.17	0.13
appendToken(Lspec/benchmarks/_202_jess/jess/Token;	17	38	2.24	17	62	0.27	0.24	0.23	0.22	0.19	0.20
findInMemory(Lspec/benchmarks/_202_jess/jess/Token	27	88	3.26	164	318	0.52	0.47	0.40	0.32	0.23	0.17
findFact(Lspec/benchmarks/_202_jess/jess/ValueVector	57	218	3.82	218	519	0.42	0.29	0.25	0.21	0.15	0.10
<init>(ILspec/benchmarks/_202_jess/jess/ValueVector;)	27	58	2.15	27	28	0.96	0.73	0.68	0.59	0.45	0.42
AddFact(Lspec/benchmarks/_202_jess/jess/ValueVector	35	78	2.23	35	89	0.39	0.33	0.32	0.30	0.25	0.25
data_equals(Lspec/benchmarks/_202_jess/jess/Token;)	24	68	2.83	21	51	0.41	0.31	0.29	0.24	0.21	0.16
equals(Lspec/benchmarks/_202_jess/jess/Value;)Z	46	158	3.43	5	7	0.71	0.83	0.71	0.50	0.38	0.25
equals(Ljava/lang/Object;)Z	37	118	3.19	5	7	0.71	0.83	0.71	0.50	0.38	0.25
FindTreeNode(Lspec/benchmarks/_205_raytrace/Point;)	97	408	4.21	101	405	0.25	0.21	0.21	0.20	0.16	0.14
Intersect(Lspec/benchmarks/_205_raytrace/Ray;Lspec/b	701	2608	3.72	178	565	0.32	0.24	0.21	0.17	0.13	0.10
<init>()V	12	28	2.33	12	8	1.50	0.80	0.75	0.63	0.55	0.41
Add(Lspec/benchmarks/_205_raytrace/Vector;)Lspec/be	23	68	2.96	23	61	0.38	0.32	0.32	0.30	0.24	0.22
Combine(Lspec/benchmarks/_205_raytrace/Point;Lspec	35	158	4.51	35	89	0.39	0.30	0.29	0.27	0.24	0.26
Set(FFF)V	10	18	1.80	10	6	1.67	1.67	1.25	1.00	0.50	0.56
Intersect(Lspec/benchmarks/_205_raytrace/Ray;Lspec/b	88	368	4.18	25	61	0.41	0.36	0.33	0.29	0.24	0.24
Check(Lspec/benchmarks/_205_raytrace/Ray;Lspec/ben	60	198	3.30	54	172	0.31	0.25	0.24	0.23	0.19	0.19
Dot(Lspec/benchmarks/_205_raytrace/Vector;)F	18	88	4.89	18	55	0.33	0.29	0.28	0.26	0.21	0.23
Ä£(Lspec/benchmarks/_222_mpegaudio/g;)Z	100	208	2.08	245	444	0.55	0.49	0.44	0.38	0.26	0.26
I([III[FI)V	74	198	2.68	84	50	1.68	1.00	0.66	0.45	0.28	0.28
J([F[F)V	305	958	3.14	305	54	5.65	3.18	2.18	1.43	0.84	0.58
read()I	16	38	2.38	12	24	0.50	0.48	0.46	0.40	0.26	0.26
read()I	59	148	2.51	34	117	0.29	0.27	0.26	0.24	0.18	0.17
Ä£(Lspec/benchmarks/_222_mpegaudio/g;)Z	133	288	2.17	43	60	0.72	0.54	0.48	0.41	0.32	0.35
Ä„([F[FII)V	34	88	2.59	277	395	0.70	0.53	0.47	0.38	0.26	0.27
e([ILspec/benchmarks/_222_mpegaudio/g;[[SI)V	59	128	2.17	53	125	0.42	0.39	0.37	0.34	0.27	0.25

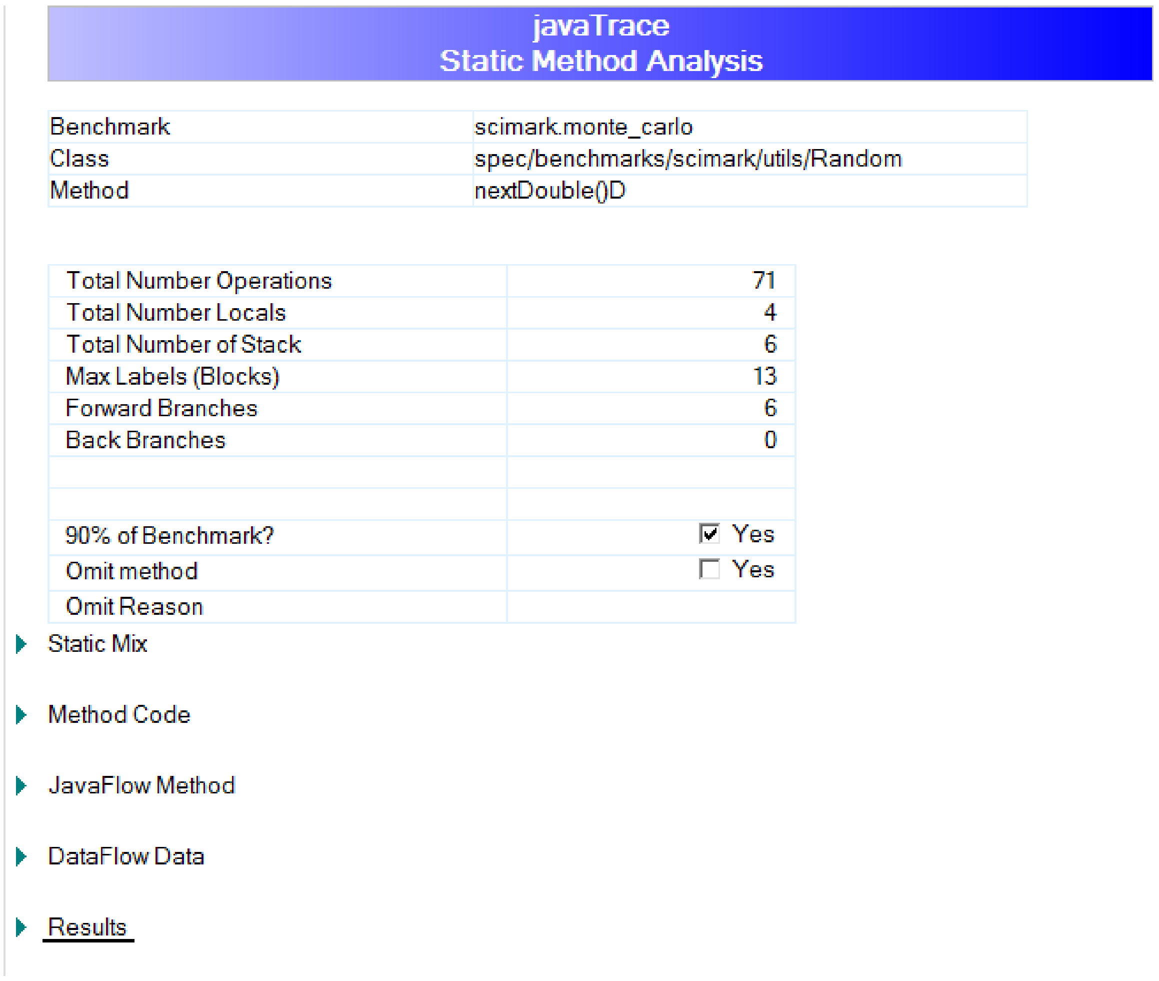
Table 45 Top 90% methods - part 4

Method	Total Inst	Hetero Nodes	N/I	Inst Exec	Mesh Cycles	IPC-0	IPC-1	IPC-2	IPC-3	IPC-4	IPC-5
j(F)S	18	68	3.78	12	51	0.24	0.24	0.24	0.22	0.17	0.16
Ä–([F[F[F)V	60	118	1.97	348	705	0.49	0.41	0.37	0.32	0.24	0.27
Ä“([F[F)V	315	958	3.04	315	97	3.25	2.27	1.74	1.25	0.79	0.58
Ä”([F[F[F)V	349	868	2.49	349	103	3.39	2.34	1.75	1.22	0.76	0.65
Ä•([F[F[F)V	649	1798	2.77	649	65	9.98	4.39	2.66	1.61	0.89	0.68
Ä£(Lspec/benchmarks/_222_mpegaudio/g;)Z	95	208	2.19	84	119	0.71	0.50	0.47	0.42	0.32	0.30
nextchar()C	168	408	2.43	44	93	0.47	0.38	0.36	0.32	0.26	0.22
readChar()C	132	278	2.11	16	54	0.30	0.26	0.25	0.25	0.20	0.21
scan_token(I)V	37	78	2.11	15	63	0.24	0.21	0.21	0.20	0.18	0.16
Move(Ljava/util/Vector;)I	41	128	3.12	212	716	0.30	0.26	0.23	0.19	0.15	0.11
AnyActiveStr([J)Z	18	48	2.67	15	47	0.32	0.26	0.24	0.21	0.15	0.15
getToken(I)Lspec/benchmarks/_228_jack/Token;	94	238	2.53	9	36	0.25	0.25	0.24	0.23	0.20	0.17
compress()V	165	398	2.41	311	571	0.54	0.42	0.36	0.28	0.19	0.17
output(I)V	208	448	2.15	88	116	0.76	0.50	0.43	0.35	0.25	0.23
decompress()V	119	338	2.84	9	13	0.69	0.69	0.64	0.53	0.38	0.22
getCode()I	164	328	2.00	139	238	0.58	0.47	0.44	0.39	0.32	0.32
pop()B	11	18	1.64	11	37	0.30	0.24	0.23	0.23	0.19	0.21
push(B)V	12	18	1.50	12	25	0.48	0.48	0.46	0.43	0.29	0.31
readByte()I	23	48	2.09	22	60	0.37	0.31	0.31	0.30	0.27	0.27
readBytes([BI)I	38	88	2.32	5	19	0.26	0.28	0.26	0.23	0.20	0.16
writeByte(B)V	12	18	1.50	12	25	0.48	0.48	0.46	0.43	0.29	0.31
updateCRC32(Ljava/util/zip/CRC32;[S)V	44	98	2.23	47	89	0.53	0.45	0.39	0.32	0.23	0.21
transform_internal([DI)V	251	608	2.42	4	18	0.22	0.20	0.15	0.11	0.11	0.09
factor([[D[I)I	162	358	2.21	47	121	0.39	0.31	0.30	0.26	0.22	0.23
nextDouble()D	71	178	2.51	49	118	0.42	0.34	0.32	0.30	0.23	0.20

APPENDIX C - SAMPLE ANALYSIS FROM ONE METHOD

This Appendix demonstrates the data gathered from one sample method, the 'nextDouble()' from the SpecJvm2008 set. This method does contribute significantly to the overall performance, and is shown as an example of the results logged for each of the methods analyzed. This data is part of a single electronic document which is part of the analysis and simulation process used to create the results presented in Chapters 5 and 7.

Figure 27 shows the summary information for the method. Both dynamic and static analysis results are summarized in this first section of the document.



javaTrace
Static Method Analysis

Benchmark	scimark.monte_carlo
Class	spec/benchmarks/scimark/utils/Random
Method	nextDouble()D

Total Number Operations	71
Total Number Locals	4
Total Number of Stack	6
Max Labels (Blocks)	13
Forward Branches	6
Back Branches	0
90% of Benchmark?	[✓ Yes]
Omit method	☐ Yes
Omit Reason	

Figure 27 Sample Analysis for nextDouble()

Figure 28 shows a section of the method document where the raw JAVAP output is stored. The method requires two columns of output from JAVAP.

```
L0:
.line 119
aload 0
getfield spec/benchmarks/scimark/utils/Random/m [I
aload 0
getfield spec/benchmarks/scimark/utils/Random/i I
iaload
aload 0
getfield spec/benchmarks/scimark/utils/Random/m [I
aload 0
getfield spec/benchmarks/scimark/utils/Random/j I
iaload
isub
istore 1
L1:
.line 120
iload 1
ifge L2
L3:
.line 121
iload 1
ldc 2147483647
iadd
istore 1
L2:
.line 122
aload 0
getfield spec/benchmarks/scimark/utils/Random/m [I
aload 0
getfield spec/benchmarks/scimark/utils/Random/j I
iload 1
iastore
L4:
.line 124
aload 0
getfield spec/benchmarks/scimark/utils/Random/i I
ifne L5
L6:
.line 125
aload 0
bipush 16
putfield spec/benchmarks/scimark/utils/Random/i I
goto L7
L5:
.line 127
aload 0
dup
getfield spec/benchmarks/scimark/utils/Random/i I
iconst_1
```

```
isub
putfield spec/benchmarks/scimark/utils/Random/i I
L7:
.line 129
aload 0
getfield spec/benchmarks/scimark/utils/Random/j I
ifne L8
L9:
.line 130
aload 0
bipush 16
putfield spec/benchmarks/scimark/utils/Random/j I
goto L10
L8:
.line 132
aload 0
dup
getfield spec/benchmarks/scimark/utils/Random/j I
iconst_1
isub
putfield spec/benchmarks/scimark/utils/Random/j I
L10:
.line 134
aload 0
getfield spec/benchmarks/scimark/utils/Random/haveRange Z
ifeq L11
L12:
.line 135
aload 0
getfield spec/benchmarks/scimark/utils/Random/left D
aload 0
getfield spec/benchmarks/scimark/utils/Random/dm1 D
iload 1
i2d
dmul
aload 0
getfield spec/benchmarks/scimark/utils/Random/width D
dmul
dadd
dreturn
L11:
.line 137
aload 0
getfield spec/benchmarks/scimark/utils/Random/dm1 D
iload 1
i2d
dmul
dreturn
L13:
.var 0 is 'this' Lspec/benchmarks/scimark/utils/Random; from L0 to L13
.var 1 is 'k' I from L1 to L13
```

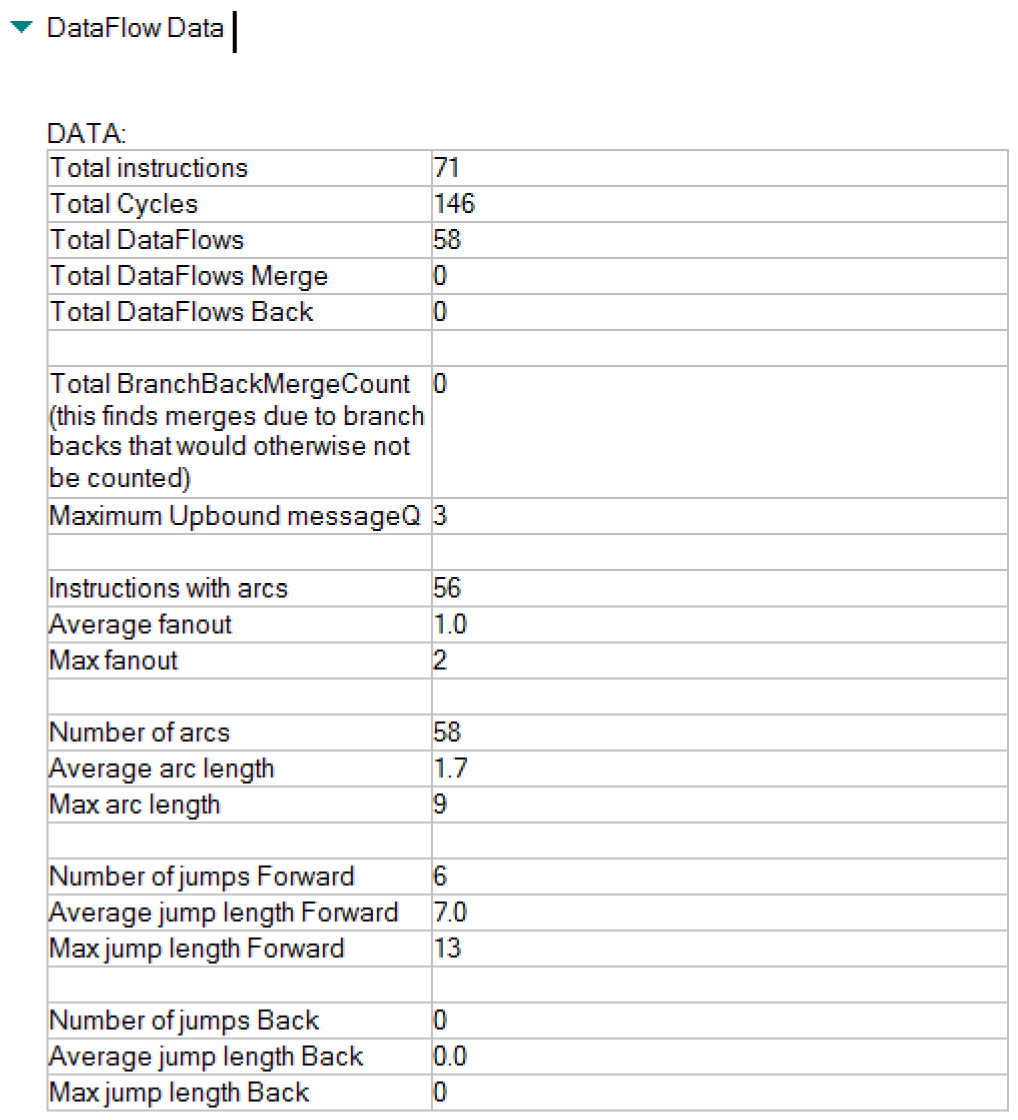
Figure 28 Method code from JAVAP – nextDouble()

Figure 30 shows the results of the DataFlow analysis applied to the method. The first entry indicates whether the instruction is jump and which direction. The data between '>>' and '<<' is the nodes to which producer data is sent. The following items indicate the number of 'pop' and 'push' values; whether the instruction loads or stores a register; and finally the instruction group.

```
Linked Code

decoder:
(0)/(+)/(-), thisAddr, Next1, Next2, Opcode,
>>Df destination. --/M-/-B/MB. Side . ThreadID;  <<
pop=n; push=n; regLoad; regStore; group
(0),0,1,,aload, >>1.--.1.0; <<pop=0;push=1;True,False,lr
(0),1,2,,getfield, >>4.--.2.0; <<pop=1;push=1;False,False,mr
(0),2,3,,aload, >>3.--.1.0; <<pop=0;push=1;True,False,lr
(0),3,4,,getfield, >>4.--.1.0; <<pop=1;push=1;False,False,mr
(0),4,5,,iaload, >>10.--.2.0; <<pop=2;push=1;False,False,mr
(0),5,6,,aload, >>6.--.1.0; <<pop=0;push=1;True,False,lr
(0),6,7,,getfield, >>9.--.2.0; <<pop=1;push=1;False,False,mr
(0),7,8,,aload, >>8.--.1.0; <<pop=0;push=1;True,False,lr
(0),8,9,,getfield, >>9.--.1.0; <<pop=1;push=1;False,False,mr
(0),9,10,,iaload, >>10.--.1.0; <<pop=2;push=1;False,False,mr
(0),10,11,,isub, >>11.--.1.0; <<pop=2;push=1;False,False,ai
(0),11,12,,istore, <<pop=1;push=0;False,True,lw
(0),12,13,,iload, >>13.--.1.0; <<pop=0;push=1;True,False,lr
(+),13,14,18,ifge, <<pop=1;push=0;False,False,jp
(0),14,15,,iload, >>16.--.2.0; <<pop=0;push=1;True,False,lr
(0),15,16,,ldc, >>16.--.1.0; <<pop=0;push=1;False,False,cn
(0),16,17,,iadd, >>17.--.1.0; <<pop=2;push=1;False,False,ai
(0),17,18,,istore, <<pop=1;push=0;False,True,lw
(0),18,19,,aload, >>19.--.1.0; <<pop=0;push=1;True,False,lr
(0),19,20,,getfield, >>23.--.3.0; <<pop=1;push=1;False,False,mr
(0),20,21,,aload, >>21.--.1.0; <<pop=0;push=1;True,False,lr
(0),21,22,,getfield, >>23.--.2.0; <<pop=1;push=1;False,False,mr
(0),22,23,,iload, >>23.--.1.0; <<pop=0;push=1;True,False,lr
(0),23,24,,iastore, <<pop=3;push=0;False,False,mw
(0),24,25,,aload, >>25.--.1.0; <<pop=0;push=1;True,False,lr
(0),25,26,,getfield, >>26.--.1.0; <<pop=1;push=1;False,False,mr
(+),26,27,31,ifne, <<pop=1;push=0;False,False,jp
(0),27,28,,aload, >>29.--.2.0; <<pop=0;push=1;True,False,lr
(0),28,29,,bipush, >>29.--.1.0; <<pop=0;push=1;False,False,mv
(0),29,30,,putfield, <<pop=2;push=0;False,False,mw
(+),30,,37,goto, <<pop=0;push=0;False,False,gt
(0),31,32,,aload, >>32.--.1.0; <<pop=0;push=1;True,False,lr
(0),32,33,,dup, >>33.--.1.0; >>36.--.2.0; <<pop=1;push=2;False,False,mv
(0),33,34,,getfield, >>35.--.2.0; <<pop=1;push=1;False,False,mr
(0),34,35,,iconst_1, >>35.--.1.0; <<pop=0;push=1;False,False,mv
(0),35,36,,isub, >>36.--.1.0; <<pop=2;push=1;False,False,ai
(0),36,37,,putfield, <<pop=2;push=0;False,False,mw
(0),37,38,,aload, >>38.--.1.0; <<pop=0;push=1;True,False,lr
(0),38,39,,getfield, >>39.--.1.0; <<pop=1;push=1;False,False,mr
(+),39,40,44,ifne, <<pop=1;push=0;False,False,jp
(0),40,41,,aload, >>42.--.2.0; <<pop=0;push=1;True,False,lr
(0),41,42,,bipush, >>42.--.1.0; <<pop=0;push=1;False,False,mv
(0),42,43,,putfield, <<pop=2;push=0;False,False,mw
(+),43,,50,goto, <<pop=0;push=0;False,False,gt
(0),44,45,,aload, >>45.--.1.0; <<pop=0;push=1;True,False,lr
(0),45,46,,dup, >>46.--.1.0; >>49.--.2.0; <<pop=1;push=2;False,False,mv
(0),46,47,,getfield, >>48.--.2.0; <<pop=1;push=1;False,False,mr
(0),47,48,,iconst_1, >>48.--.1.0; <<pop=0;push=1;False,False,mv
(0),48,49,,isub, >>49.--.1.0; <<pop=2;push=1;False,False,ai
(0),49,50,,putfield, <<pop=2;push=0;False,False,mw
(0),50,51,,aload, >>51.--.1.0; <<pop=0;push=1;True,False,lr
(0),51,52,,getfield, >>52.--.1.0; <<pop=1;push=1;False,False,mr
(+),52,53,65,ifeq, <<pop=1;push=0;False,False,jp
(0),53,54,,aload, >>54.--.1.0; <<pop=0;push=1;True,False,lr
(0),54,55,,getfield, >>63.--.2.0; <<pop=1;push=1;False,False,mr
(0),55,56,,aload, >>56.--.1.0; <<pop=0;push=1;True,False,lr
(0),56,57,,getfield, >>59.--.2.0; <<pop=1;push=1;False,False,mr
(0),57,58,,iload, >>58.--.1.0; <<pop=0;push=1;True,False,lr
(0),58,59,,i2d, >>59.--.1.0; <<pop=1;push=1;False,False,ci
(0),59,60,,dmul, >>62.--.2.0; <<pop=2;push=1;False,False,af
(0),60,61,,aload, >>61.--.1.0; <<pop=0;push=1;True,False,lr
(0),61,62,,getfield, >>62.--.1.0; <<pop=1;push=1;False,False,mr
(0),62,63,,dmul, >>63.--.1.0; <<pop=2;push=1;False,False,af
(0),63,64,,dadd, >>64.--.1.0; <<pop=2;push=1;False,False,af
(0),64,,,dreturn, <<pop=1;push=0;False,False,rt
(0),65,66,,aload, >>66.--.1.0; <<pop=0;push=1;True,False,lr
(0),66,67,,getfield, >>69.--.2.0; <<pop=1;push=1;False,False,mr
(0),67,68,,iload, >>68.--.1.0; <<pop=0;push=1;True,False,lr
(0),68,69,,i2d, >>69.--.1.0; <<pop=1;push=1;False,False,ci
(0),69,70,,dmul, >>70.--.1.0; <<pop=2;push=1;False,False,af
(0),70,,,dreturn, <<pop=1;push=0;False,False,rt
```

Figure 29 DataFlow code - nextDouble()

Figure 31 shows the raw data from the results of the dataflow analysis. The items were described in Section 7.2, and are presented for a single method as a demonstration of the data gathering processes.



DATA:

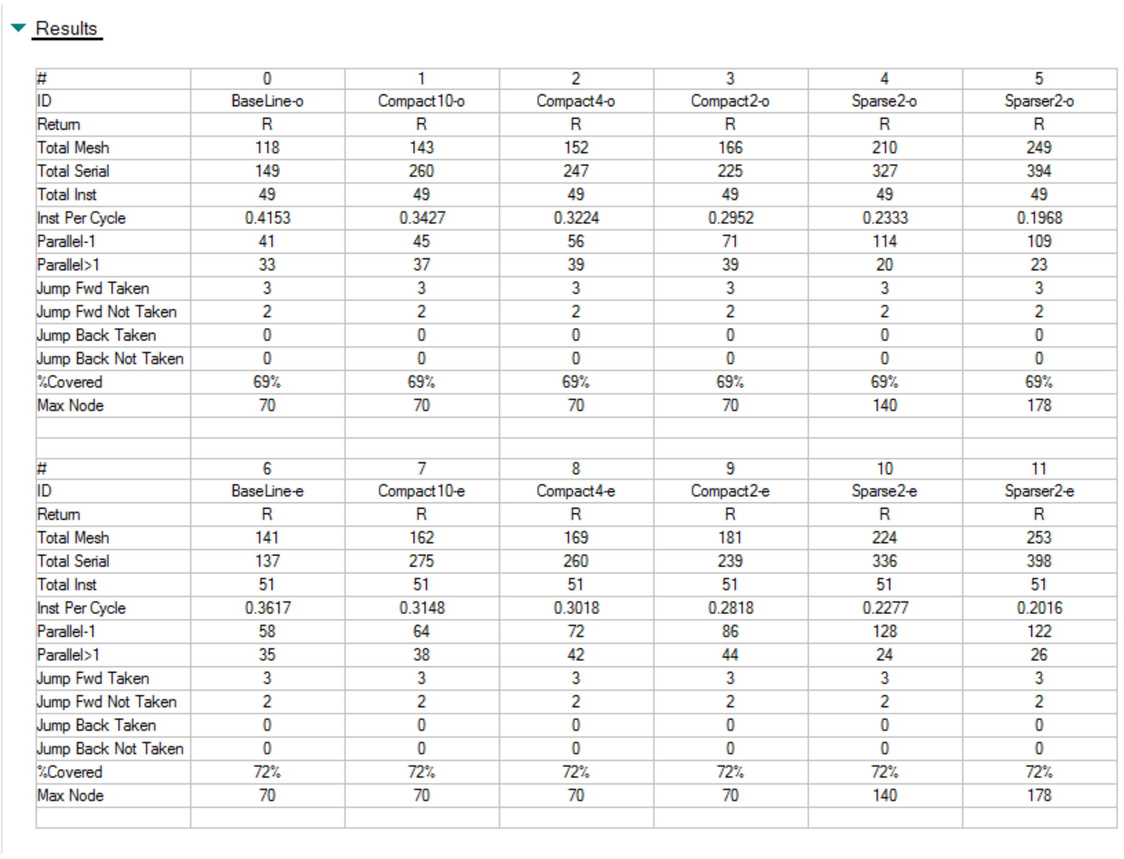
Total instructions	71
Total Cycles	146
Total DataFlows	58
Total DataFlows Merge	0
Total DataFlows Back	0
Total BranchBackMergeCount (this finds merges due to branch backs that would otherwise not be counted)	0
Maximum Upbound messageQ	3
Instructions with arcs	56
Average fanout	1.0
Max fanout	2
Number of arcs	58
Average arc length	1.7
Max arc length	9
Number of jumps Forward	6
Average jump length Forward	7.0
Max jump length Forward	13
Number of jumps Back	0
Average jump length Back	0.0
Max jump length Back	0

Figure 30 DataFlow Analysis - nextDouble()

Figure 31 represents the raw data from this single method in the performance analysis presented in Section 7.3. The two sections of the table represent the 2 Branch Prediction cases simulated. The columns represent the 6 configurations simulated. The rows represent data as follows:

- The 'R' in the Return column indicates that at Return instruction was executed.
- Total mesh clock cycles required to execute the method
- Total serial clock cycles required to execute the method
- Total instructions executed
- Instructions per cycle which is the ratio of instructions to mesh clock cycles
- The Parallel 1 row indicates the number of mesh cycles where 2 instructions were executing
- Parallel>1 indicates the number of mesh cycles where more than 2 instructions were executing
- Jump Forward Taken
- Jump Forward Not Taken
- Jump Backward Taken (Note that this method although contributing to the overall performance does not have a loop inside. This method would benefit due to its residence in the DataFlow Fabric and being invoked repeatedly by the processes described in Section 6.2.
- Jump Backward Not Taken
- %Coverage is the ratio of static instructions executed to total number of static instructions. This is the test of viability of the jump/branch prediction strategy described in Section 7.1.

- Max Nodes represents the serial node required by the last instruction in the method. This number is static in 5 of the cases, and is only interesting as instructions are deployed to a heterogeneous DataFlow Fabric in Case 5. Note that this is not the number of nodes consumed, but only represents the added distance that data might have to traverse in the execution of the method.


Results

#	0	1	2	3	4	5
ID	BaseLine-o	Compact10-o	Compact4-o	Compact2-o	Sparse2-o	Sparser2-o
Return	R	R	R	R	R	R
Total Mesh	118	143	152	166	210	249
Total Serial	149	260	247	225	327	394
Total Inst	49	49	49	49	49	49
Inst Per Cycle	0.4153	0.3427	0.3224	0.2952	0.2333	0.1968
Parallel-1	41	45	56	71	114	109
Parallel>1	33	37	39	39	20	23
Jump Fwd Taken	3	3	3	3	3	3
Jump Fwd Not Taken	2	2	2	2	2	2
Jump Back Taken	0	0	0	0	0	0
Jump Back Not Taken	0	0	0	0	0	0
%Covered	69%	69%	69%	69%	69%	69%
Max Node	70	70	70	70	140	178

#	6	7	8	9	10	11
ID	BaseLine-e	Compact10-e	Compact4-e	Compact2-e	Sparse2-e	Sparser2-e
Return	R	R	R	R	R	R
Total Mesh	141	162	169	181	224	253
Total Serial	137	275	260	239	336	398
Total Inst	51	51	51	51	51	51
Inst Per Cycle	0.3617	0.3148	0.3018	0.2818	0.2277	0.2016
Parallel-1	58	64	72	86	128	122
Parallel>1	35	38	42	44	24	26
Jump Fwd Taken	3	3	3	3	3	3
Jump Fwd Not Taken	2	2	2	2	2	2
Jump Back Taken	0	0	0	0	0	0
Jump Back Not Taken	0	0	0	0	0	0
%Covered	72%	72%	72%	72%	72%	72%
Max Node	70	70	70	70	140	178

Figure 31 Simulation results - nextDouble()

APPENDIX D - BENCHMARK DESCRIPTIONS

Appendix D provides a list of all the benchmarks that are included in the SpecJvm2008 and SpecJvm98 suites. Benchmarks were excluded due to being part of the initialization and due to execution difficulties. The required use of the GNU Classpath [52] code rather than the more recently released (open source) Oracle Java classes is the primary reason for these exclusions on SpecJvm2008. Table 46 shows SpecJvm98 benchmarks both included and excluded. Table 47 shows SpecJvm2008 benchmarks included in the analysis; while Table 48 shows SpecJvm2008 benchmarks that were excluded.

Table 46 SpecJvm98 Benchmarks

- SpecJvm98- Included
    - _201_compress
        - A popular utility used to compress/uncompress files
    - _202_jess
        - A Java expert system shell
    - _209_db
        - A small data management program
    - _222_mpegaudio
        - An MPEG-3 audio stream decoder
    - _227_mtrt
        - A dual-threaded program that ray traces an image file
    - _228_jack
        - A parser generator with lexical analysis
- SpecJvm98- Excluded
    - _200_check
        - checks JVM and Java features
    - _213_javac
        - The Java compiler, compiling 225,000 lines of code

Table 47 SpecJvm2008 Benchmarks Included

SpecJvm2008- Included

- Compress
    - This benchmark compresses data, using a modified Lempel-Ziv method (LZW). Basically finds common substrings and replaces them with a variable size code.
- Crypto
    - This benchmark focuses on different areas of crypto and are split in three different sub-benchmarks.
    - signverify
        - Sign and verify using MD5withRSA, SHA1withRSA, SHA1withDSA and SHA256withRSA protocols.
- MPEGaudio
    - This benchmark is very similar to the SPECjvm98 mpegaudio. The mp3 library has been replaced with JLayer, an LGPL mp3 library. Its floating-point heavy and a good test of mp3 decoding. Input data were taken from SPECjvm98.
- Scimark
    - This benchmark was developed by NIST and is widely used by the industry as a floating point benchmark.
    - FFT
        - Fast Fourier Transform (FFT) performs a one-dimensional forward transform of 4K complex numbers.
    - SOR
        - Jacobi Successive Over-relaxation (SOR) on a 100x100 grid exercises typical access patterns in finite difference applications, for example, solving Laplace's equation in 2D with Drichlet boundary conditions.
    - Monte Carlo
        - Monte Carlo integration approximates the value of Pi by computing the integral of the quarter circle y = sqrt(1 - x^2) on [0,1].
    - Sparse
        - Sparse matrix multiply uses an unstructured sparse matrix stored in compressed-row format with a prescribed sparsity structure.
    - LU
        - dense LU matrix factorization Computes the LU factorization of a dense 100x100 matrix using partial pivoting.

Table 48 SpecJvm2008 Benchmarks Excluded

SpecJvm2008- Excluded	
Startup	This benchmark starts each benchmark for one operation.
Compiler	This benchmark uses the OpenJDK (JDK 7 alpha) front end compiler to compile a set of .java files.
Crypto	This benchmark focuses on different areas of crypto and are split in three different sub-benchmarks.
aes	Encrypt and decrypt using the AES and DES protocols, using CBC/PKCS5 Padding and CBC/NoPadding
rsa	Encrypt and decrypt using the RSA protocol, using input data of size 100 bytes and 16 kB.
Derby	This benchmark uses an open-source database written in pure Java. It is synthesized with business logic to stress the BigDecimal library. It is a direct replacement to the SPECjvm98 db benchmark.
Serial	This benchmark serializes and deserializes primitives and objects, using data from the JBoss benchmark.
Sunflow	This benchmark tests graphics visualization using an open source, internally multi-threaded global illumination rendering system.
XML	This benchmark has two sub-benchmarks: XML.transform and XML.validation.

Glossary

ACK Protocol.	Synchronizing the transmission of data between two elements by sending an 'Acknowledgement' message which must be received before any further data transmission.
Anchor Node.	A special Instruction Node in the DataFlow Fabric which acts as the interface between the Fabric and the General Purpose Processor. Method state is kept, and this node may allow invocation from another node in the DataFlow Fabric.
Branch Prediction.	The process of predicting the behavior of branches to improve performance by avoiding stalls in the instruction stream. This function is not part of the JavaFlow machine.
ByteCode.	See Java ByteCode
Class.	The basic unit of programming in the Java language. Classes may be used statically or instantiated into one more many objects which form the basis for the Object Oriented Design of the Java language.
Class File.	The basic unit of information containing all the information about a Java Class for it to be loaded, linked, and executed in a Java Virtual Machine.
Constant Pool.	A portion of the Java ClassFile which contains references to other classes used by the current class.

	This data is updated upon loading of the class to resolve these references to actual pointers inside the JVM.
ControlFlow.	The normal ordering of a program/method. Usually associated with an instruction counter which indicates which instruction is currently executing.
Dimensional Routing.	The process of routing data in the DataFlow Fabric that always routes in first one dimension (x) and then the second dimension (y). This routing algorithm while not as complex or optimal as some allows a simple router, sequential delivery, and no deadlocks.
DataFlow.	The ordering of execution of a program/method based on the availability of data for the operations. Note that this DataFlow ordering may contradict the intended control flow ordering in many languages and requires special handling to execute programs properly.
Dynamic Mix.	The percentages of instructions actually executed during the running of a benchmark.
Fabric.	The name for the set of Instruction Nodes which are interconnected by the mesh network in a DataFlow machine.
Field.	A variable of a Class or an Object instantiated from a class. This provides a structured way to view only the

	data the designer wants to be made available outside the construct of a Class.
Finally clause.	A Java language construct that executes a set of instructions after a method is possibly an interrupted by an exception.
Fire.	The action of an Instruction Node actually executing the instruction. Normally this is initiated by the receipt of all necessary data elements (e.g. ‘Pop’==’PopsReceived’)
Folding.	The process of removing possibly redundant ByteCode instructions from a method.
Garbage Collection.	The freeing of storage used by objects that are no longer necessary
General Purpose Process (GPP).	A standard processor core capable of interpreting Java ByteCode instructions and managing the DataFlow Fabric.
Graphics Processing Unit (GPU)	The processing units used in advanced graphics subsystems and also exploited for advanced parallel computing applications. These processors have very high level of parallelism, but due to their SIMD (Single Instruction, Multiple Data) organization, and typical restrictive cross processor memory access, are difficult to program for general purpose functions.

Heap.	The set of memory addresses used by the Java Virtual Machine to store objects. The Heap is typically subject to Garbage Collection.
Hyperblock.	A subset of a program in the TRIPS context that can execute atomically and not have any back branches.
Instance.	An object created (instantiated) from a Class. Note that instance/object data is maintained on the Heap while static Class data is maintained in a common Method Area.
InstanceID.	A field in messages to insure that the message is received by nodes that are part of the same Thread, Class, and method. In a practical implementation, this value would be hashed when included in messages.
Instruction Node.	An element in the DataFlow Fabric responsible for the execution of a ByteCode Instruction. Included in the Instruction Node is the instruction execution unit, one or more instruction data units, serial network router, mesh network router, and at times, the GPP/Memory interface unit.
Instructions Per Cycle (IPC). .	The performance measurement of the system. The cycles are the Mesh Node cycles, and the instructions are the executed Java ByteCode instructions. This metric is used as it is independent of both the technology and (to a degree), the exact execution times of the instructions.

Java ByteCode	The instruction set architecture (ISA) implemented by the Java Virtual Machine
Java Virtual Machine (JVM). . . .	The architecture including instruction set, machine resources, and interfaces of the processing system used to execute Java programs.
JAVAC.	The Java compiler which compiles source java classes into Java ClassFiles for loading and executing by the Java Virtual Machine.
JavaFlow.	The single word name for the Java DataFlow Machine.
JAVAP.	A program provided as part of the Java Development environment to disassemble Java Class Files into human readable statements.
Linear Address.	The addresses of the ByteCode instructions prior to loading into the DataFlow Fabric. While the architecture specifies the addresses of the instructions as byte offsets from the beginning, the JavaFlow machine linear addresses are sequential for each instruction.
Locality.	The concept of insuring that the data required by a processing unit is physically close to that unit.
Mesh Address.	Addresses of cells in the DataFlow Fabric. In the JavaFlow machine these addresses are 3 tuple: x, y, p where x and y represent the grid positions, and p is the

	processing function. In the JavaFlow analysis, p is always set to 0.
Mesh Clock cycle.	The time for the instruction execution unit to perform a portion of the execution and the time for the Mesh Router to transfer data through the Instruction Node.
Mesh Message.	The packet of data transferred from one Instruction Node to another using the mesh network.
Method.	The program which executes as part of a Java Class
Method Area.	An area of storage where the method code, associated data, and static class data is stored.
Object.	An instantiation of a Class. The object has storage allocated on the Heap.
Ordered Memory Access.	Memory accesses to the Heap where the memory subsystem must insure the proper ordering of reads and writes to insure proper sequencing.
Parallelism.	Execution events occurring at the same time. In JavaFlow, parallelism is measured by the number of mesh cycles that have more than one Instruction Node executing. Note that this includes actual processing, but excludes service times.
Pop.	A characteristic of a ByteCode instruction that defines the number of stack data elements that are used as input to the instruction.

PopsReceived.	A state in the Instruction Node that counts the number of DataFlow messages received. When 'PopsReceived' equals 'Pop,' the instruction is ready to fire.
Producer-Consumer.	The nomenclature of a DataFlow machine where Instruction Nodes produce the results of computations and then send these results to consuming nodes elsewhere in the DataFlow Fabric.
Push.	A characteristic of a ByteCode instruction that defines the number of destinations to which the results must be sent.
Quick.	A term applied to the opcodes of ByteCode instructions whose addresses have been resolved. While not part of the official JVM architecture, these opcodes have been used by interpreters as a pseudo standard.
Serial Clock cycle.	The clock used to transmit data down the serial network. It is projected that this clock cycle can be shorter than the mesh network due to the less complex routing functions required.
Serial Message.	The message sent between nodes in the serial network which is used to maintain control flow ordering and to transfer register data to ByteCode instructions.

Simultaneous Multi-Threading. .	A processing function implemented in some modern systems where more than one thread can be executing in a single set of processing elements by maintaining instance id information along with data elements.
SPEC.	Standard Performance Evaluation Corporation is a non-profit corporation formed to standardize and support a series of benchmarks used measure the performance of computing systems.
Stack.	A resource of the JVM which holds operands for ByteCode instructions. The Stack is a push down (Last In, First Out) structure.
Static Mix.	The percentages of instructions found in methods that are loaded into the machine independent of their execution frequency.
Unordered Memory Access.	Memory accesses to the Constant Pool which since the resolution of addresses is done before loading instructions, can be made without the constraints of insuring ordering the accesses between Instruction Nodes.
X-Y Routing.	See Dimensional Routing

References

[1] P. Miller, *The Smart Swarm*, New York, NY: Avery Books, 2010.

[2] TIOBE Software BV. (2007, October 12, 2007). *TIOBE Programming Community Index*. Available at: http://www.tiobe.com/index.php

[3] S. Cass, "The Top 10 Programming Languages; Spectrum's 2014 Rankings," *IEEE Spectrum,* vol. 51, issue 7, p. 68, 2014.

[4] T. Lindholm and F. Yellin, *Java Virtual Machine Specification*, Boston, MA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[5] B. Venners, *Inside the Java Virtual Machine*, New York: McGraw-Hill Professional, 1999.

[6] R. J. Ascott and E. E. Swartzlander Jr., "JavaFlow—A Java dataflow machine," in *IEEE International SOC Conference*, Belfast, Northern Ireland, UK, pp. 211-214, 2009.

[7] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin, "Dataflow: The Road Less Complex," presented at the In the Workshop on Complexity-effective Design (WCED) held in conjunction with the 30th Annual International Symposium on Computer Architecture (ISCA), 2003.

[8] R. Ho, K. W. Mai, and M. A. Horowitz, "The future of wires," *Proceedings of the IEEE,* vol. 89, issue 4, pp. 490-504, 2001.

[9] P. Kongetira, K. Aingaran, and K. Olukotun, "Niagara: a 32-way multithreaded Sparc processor," *IEEE Micro,* vol. 25, issue 2, pp. 21-29, 2005.

[10] D. Nicolaescu and A. Veidenbaum, "Understanding and comparing the performance of optimized JVMs," in *Innovative Architecture for Future Generation High-Performance Processors and Systems*, pp. 1-8, 2005.

[11] K. Casey, M. A. Ertl, and D. Gregg, "Optimizing indirect branch prediction accuracy in virtual machine interpreters," *ACM Trans. Program. Lang. Syst.,* vol. 29, issue 6, pp. 37:1-37:29, 2007.

[12] C. Click, G. Tene, and M. Wolf, "The pauseless GC algorithm," *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments,* pp. 46-56, 2005.

[13] D. Greve, "Symbolic simulation of the JEM1 microprocessor," in *Formal Methods in Computer-Aided Design*. vol. 1522, 1st ed., Berlin: Springer, 1998, p. 531.

[14] J. M. O'Connor and M. Tremblay, "picoJava-I: The Java Virtual Machine in hardware," *IEEE Micro,* vol. 17, issue 2, pp. 45-53, 1997.

[15] M. Schoeberl, "JOP: A Java Optimized Processor for Embedded Real-Time Systems," Ph.D. Dissertation, Vienna University of Technology, Vienna, 2005.

[16] M. Schoeberl, "Evaluation of a Java processor," *Tagungsband Austrochip 2005,* pp. 127-134, October, 2005.

[17] J. C. B. Mattos, S. Wong, and L. Carro, "The Molen FemtoJava engine," in *Proceedings of the IEEE 17th International Conference on Application-Specific Systems, Architectures and Processors*, pp. 19-22, 2006.

[18] R. Radhakrishnan, "Microarchitectural Techniques to Enable Efficient Java Execution," Ph.D. Dissertation, University of Texas at Austin, Austin, TX, 2000.

[19] R. Radhakrishnan, R. Bhargava, and L. K. John, "Improving Java performance using hardware translation," in *15th International Conference on Supercomputing*, pp. 427-439, 2001.

[20] J. Glossner and S. Vassiliadis, "Delft-Java Dynamic Translation," in *25th EUROMICRO Conference*, Milan, Italy, pp. 57-62, 1999.

[21] H. Oi, "Instruction Folding in a Hardware-Translation Based Java Vvirtual Machine," in *3rd Conference on Computing Frontiers*, pp. 139-146, 2006.

[22] M. W. El-Kharashi, F. Elguibaly, and K. F. Li, "Adapting Tomasulo's algorithm for bytecode folding based Java processors," *ACM SIGARCH Computer Architecture News,* vol. 29, issue 5, pp. 1-8, 2001.

[23] H. C. Wang and C. K. Yuen, "Exploiting Dataflow to Extract Java Instruction Level Parallelism on a Tag-Based Multi-Issue Semi In-order (TMSI) Processor," in *20th International Parallel and Distributed Processing Symposium*, pp. 9-17, 2006.

[24] N. Vijaykrishnan, N. Ranganathan, and R. Gadekarla, "Object-oriented architectural support for a Java processor," *ECOOP'98—Object-Oriented Programming,* pp. 330-355, 1998.

[25] Adapteva, Inc.,. (2013, July, 2014). *Epiphany Multicore IP*. Available at: http://www.adapteva.com/

[26] HSA Foundation. (2014). *Heterogeneous System Architecture*. Available at: http://www.hsafoundation.com/

[27] Oracle Corporation. (2014). *Java 8 Documentation*. Available at: http://www.oracle.com/technetwork/java/javase/documentation/index.html

[28] OpenJDK. (2014). *OpenJDK Sumatra Project*. Available at: http://wiki.openjdk.java.net/display/Sumatra/Main

[29] J. Hsu, "IBM's New Brain," *IEEE Spectrum,* vol. 51, issue 10, pp. 17-19, 2014.

[30] J. B. Dennis and D. P. Misunas, "A preliminary architecture for a basic data-flow processor," *ACM SIGARCH Computer Architecture News,* vol. 3, issue 4, pp. 126-132, 1975.

[31] J. Sharp, *Data Flow Computing: Theory and Practice*, Norwood, NJ: Ablex Pub, 1992.

[32] D. A. Adams, "A Computation Model With Data Flow Sequencing," Stanford University Technical Report, Palo Alto, CA., 1968

[33] L. C. Hobbs, "Parallel Processor Systems, Technologies, and Applications," in *A Model for Parallel Computations*, D. A. Adams, Ed., 1st ed., New York: Spartan, 1970, pp. 311-333.

[34] J. Rumbaugh, "A Data Flow Multiprocessor," *IEEE Transactions on Computers,* vol. C-26, issue 2, pp. 138-146, 1977.

[35] S. Swanson, A. Schwerin, M. Mercaldi, A. Petersen, A. Putnam, K. Michelson*, et al.*, "The WaveScalar architecture," *ACM Transactions on Computer Systems,* vol. 25, issue 2, pp. 1-54, 2007.

[36] A. Smith, R. Nagarajan, K. Sankaralingam, R. McDonald, D. Burger, S. W. Keckler*, et al.*, "Dataflow predication," *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture,* pp. 89-102, 2006.

[37] M. Beck, R. Johnson, and K. Pingali, "From Control Flow to Dataflow.," Cornell University, Technical Report TR89-1050, 1989.

[38] G. M. Papadopoulos and D. E. Culler, "Monsoon: an Explicit Token-Store Architecture," *ACM SIGARCH Computer Architecture News,* vol. 18, issue 3, pp. 82-91, 1990.

[39] K. R. Traub, "A compiler for the MIT Tagged-Token dataflow architecture," MIT Technical Report, Cambridge, MA., 1986.

[40] J. R. Gurd, C. C. Kirkham, and I. Watson, "The Manchester prototype dataflow computer," *Communications of the ACM,* vol. 28, issue 1, pp. 34-52, 1985.

[41] D. Burger, S. W. Keckler, K. S. McKinley, M. Dahlin, L. K. John, C. Lin*, et al.*, "Scaling to the End of Silicon with EDGE Architectures," *Computer,* vol. 37, issue 7, pp. 44-55, 2004.

[42] R. McDonald, D. Burger, S. W. Keckler, K. Sankaralingam, and R. Nagarajan, "Trips processor reference manual," Technical report, Department of Computer Sciences, The University of Texas at Austin, 2005.

[43] A. Smith, J. Burrill, J. Gibson, B. Maher, N. Nethercote, B. Yoder*, et al.*, "Compiling for EDGE architectures," *International Symposium on Code Generation and Optimization,* pp. 185–195, 2006.

[44] M. Gebhart, B. A. Maher, K. E. Coons, J. Diamond, P. Gratz, M. Marino*, et al.*, "An evaluation of the TRIPS computer system," *ACM SIGPLAN Notices,* vol. 44, issue 3, pp. 1-12, 2009.

[45] S. Swanson, A. Putnam, M. Mercaldi, K. Michelson, A. Petersen, A. Schwerin*, et al.*, "Area-performance trade-offs in tiled dataflow architectures," in *33rd International Symposium on Computer Architecture*, pp. 314-326, 2006.

[46] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin, "WaveScalar," *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture,* pp. 291–302, 2003.

[47] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley, *The Java Virtual Machine Specification, Java SE 8 Edition*, Upper Saddle River, NJ: Addison-Wesley, 2014.

[48] M. N. Horak, S. M. Nowick, M. Carlberg, and U. Vishkin, "A Low-Overhead Asynchronous Interconnection Network for GALS Chip Multiprocessors," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* vol. 30, issue 4, pp. 494-507, 2011.

[49] A. Sheibanyrad, A. Greiner, and I. Miro-Panades, "Multisynchronous and Fully Asynchronous NoCs for GALS Architectures," *IEEE Design & Test of Computers,* vol. 25, issue 6, pp. 572-580, 2008.

[50] SPEC. (2010). *Standard Performance Evaluation Corporation*. Available at: www.spec.org

[51] R. Lougher. (2010, August, 2014). *JAMVM*. Available at: http://jamvm.sourceforge.net/

[52] Free Software Foundation, Inc.,. (2009). *GNU Classpath*. Available at: http://www.gnu.org/software/classpath/classpath.html

[53] R. Radhakrishnan, N. Vijaykrishnan, L. K. John, A. Sivasubramaniam, J. Rubio, and J. Sabarinathan, "Java runtime systems: characterization and architectural implications," *IEEE Transactions on Computers,* vol. 50, issue 2, pp. 131-146, 2001.

[54] M. Dahm, "Technical Report B-17-98 Byte code engineering with the BCEL API," Institut fur Informatik Freie Universit at Berlin, 2001.

[55] OW2_Consortium. (2010). *ASM Documentation*. Available at: http://asm.ow2.org

[56] J. Meyer and T. Downing, *Java Virtual Machine*, Sebastopol, CA: O'Reilly & Associates, 1997.