Peppermint and *Sled*: Tools for Evaluating SMP Systems based on IA-64 (IPF) Processors

Sujoy Basu*, Sumit Roy and Raj Kumar
Hewlett-Packard Laboratories
Hewlett-Packard Company
1501 Page Mill Rd.
Palo Alto, CA 94304

Tom Fisher and Bruce E. Blaho
Technical Computing Division
Hewlett-Packard Company
3404 E Harmony Rd.
Fort Collins, CO 80528

Abstract

In this paper, we describe Peppermint *and* Sled*: tools developed for evaluations of computer systems based on IA-64 processors.* Sled *generates trace from applications running on IA-64 processors, while* Peppermint *models the complete system using cycle-accurate, trace-driven simulation.* Peppermint *is based on* Augmint *[9], which leaves open the possibility of doing execution-driven simulations in future.*

Peppermint *and* Sled *allow us to perform a trace-based evaluation of 4 applications running on SMP systems based on* Itanium *and* McKinley *processors. We find that the improvement in IPC of* McKinley *relative to* Itanium *ranges from 7% to over 100% for our different applications. The improvement can be attributed to a variety of factors. These range from the availability of additional functional units and issue ports in the* McKinley *processor to our assumption of a better memory system. While the improvement in performance remains valid in SMP systems in some cases, higher contention for system bus and memory reduces the performance gain in other cases. Increasing the system bus bandwidth and size of queues for pending requests in the memory controller are identified as first steps for optimizing SMP performance.*

1 Introduction

In this paper, we describe *Peppermint* and *Sled*: tools developed for cycle-accurate, trace-driven simulation of applications running on IA-64 processors, with the possibility of doing execution-driven simulations in future. With the introduction of *Itanium* processor[10], it has become necessary for system architects to understand the performance of applications developed for the IA-64 family of processors. They would like to spend their time on those aspects of the design where the maximum impact on performance could be realized. Some of the different scenarios in which we envisage the use of *Peppermint* and *Sled* are illustrated in Figure 1.

The design of our simulator *Peppermint* is influenced by the complex, real-world technical applications we need to analyze. The tools will be used primarily to study long traces captured from such applications, and simulate the traces by varying various parameters of system components like the memory controller. However as shown in our experiments in this paper, *Peppermint* can be used to explore other aspects of the design space. The motivation for this project has been the development of a tool set that can be used for fast exploration of the design space for systems built around IA-64 processors. Detailed performance simulations, such as those possible using using flexible simulation environments like [3], can be very slow. Because of our requirement for fast simulations, we have chosen to selectively model system components. We might iteratively refine the model later if higher accuracy is required at the cost of slower simulation speed.

Peppermint currently models one or more IA-64 processor with three levels of cache, a system bus, memory controller and DRAM chips. All these components are parameterized. *Peppermint* uses a configuration file in which these parameters are specified. The focus of this paper is to give the maximum amount of information about the details of our implementation. Since we are selectively modeling details of the system architecture, we have chosen to highlight these details. Then we give the reader some idea about the basic experiments that can be conducted using our tools.

Although the target architecture for *Peppermint* is an IA-64 system, only the generation of IA-64 traces have been done on *Itanium* systems using *Sled* . *Peppermint* itself has been developed and tested primarily on PA-RISC computers running HP-UX and IA-32 computers running Linux. It takes the trace and a configuration file as input. The actual

* contact at *sujoy_basu@hp.com*

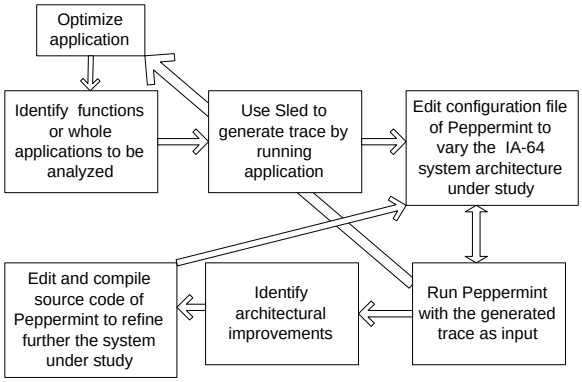

Figure 1. Possible uses for *Peppermint* and *Sled*

application or the libraries used by it on the IA-64 system need not be provided to *Peppermint* since *Sled* captures all the necessary information in the trace.

This paper is organized as follows. Section 2 describes how *Peppermint* models different components of the system architecture, and our ongoing work on validation. Section 3 describes in detail how *Sled* , the tool for capturing IA-64 application traces, is implemented and performs.

2 Architectural Model

In this section, we describe the details of the system architecture modeled by *Peppermint*. In case the reader is interested in extending the model, experience in writing architectural models for *Augmint* [9] will be required.

2.1 Processor Architecture

2.1.1 Instruction Dispersal

Peppermint parameterizes the issue width of the processor. For both *Itanium* and *McKinley*, this number is 2. This means that at most 2 instruction bundles will be available for dispersal every cycle in the *dispersal window*. Experiments can be conducted by varying this parameter in *Peppermint*. Dispersal is achieved with a decoupling buffer, which decouples the front end of the pipeline from the rest of the pipeline. The front end is responsible for fetching instructions from instruction cache into the decoupling buffer, while the back end disperses them. *Peppermint* has one primary event corresponding to the front end, and another primary event corresponding to the back end. Both of these events are scheduled on every clock cycle.

The event corresponding to the front end reads entries from the trace file, and schedules corresponding instruction fetches. The number of such instruction fetches in a cycle may be limited, of course, by factors like available ports on the I-cache and number of outstanding cache misses. These instruction fetch events will in some subsequent clock cycle feed the corresponding instruction bundle into the decoupling buffer. Here perfect branch prediction is assumed. The model can be extended to introduce delay due to branch mispredictions.

The event corresponding to the back end starts by checking that the dispersal window is filled with bundles. If not, it shifts the next bundles in the instruction stream from the decoupling buffer into the dispersal window. This succeeds unless the bundle is not available due to cache miss. Next it tries to issue all the instructions in the dispersal window to their issue ports within the same cycle. To do that, the instruction bundle is decoded using functions from a library used by linker and other system tools. Next, rules available in [7] are used to determine which issue port can be given an instruction. As soon as an instruction fails to find a free issue port, dispersal stops for that cycle. All instructions in sequential order preceding that instruction have been dispersed that cycle. Dispersal is said to split-issue at that instruction. Also, an explicit stop-bit might be encoded in the bundle, indicating split-issue at an instruction. Other special cases for *Itanium* that cause split-issue are implemented. The reader is referred to [7] for these details. Another important cause for split-issue that we implement is an instruction waiting for a source operand register that is the target of a load from memory.

2.1.2 Predication

Predicated instructions, for which the predicate register contains 0, must be squashed in the pipeline without updating the architectural state. Specifically, *Peppermint* cannot mark the target register of such an instruction busy, forcing some instruction to wait for the register to become available. This is currently modeled when the predicated instruction is a *load* or *store*. *Peppermint* uses the value of the predicate

register recorded by *Sled* in the *load* or *store* trace entry to determine whether the load or store needs to be issued to the memory system, and also whether the target register of a load needs to be marked busy. This is quite good as a first-order approximation since loads can have high latency due to cache misses. If higher accuracy of simulation is desired, the value of predicate register has to be recorded in the trace for all instructions; then *Peppermint* can ensure that other predicated-off instructions do not mark their target register busy.

2.1.3 Data Speculation

Data speculation is a technique that allows the compiler to schedule loads ahead of time. In this case, though, the speculation arises from the fact that the load has been scheduled prior to stores that actually precede it in program order and can potentially write into the memory location from which the load will be reading. Compile-time analysis for ruling out such overlapping stores and loads is difficult when dynamic data structures (code with pointers) preclude static analysis. The IA-64 architecture supports such data speculation by introducing *advance load* instructions. An advance load behaves like a normal load in accessing memory. However, in addition to that, the target register, load address and number of bytes being loaded are entered in the Advance Load Address Table (ALAT). Every store instruction checks the ALAT for entries with overlapping addresses. Such entries are invalidated. At the original location of the load in the instruction stream, a speculation check (*chk.a*) instruction is placed. When executed, it checks the ALAT for the entry inserted by the corresponding advance load. If the entry is present, speculation has succeeded. Otherwise it has failed, and a branch is taken to fix-up code.

Peppermint currently does not model the ALAT. However it does issue advance loads to memory, which is what the architecture specifies. Notice also that during the capture of a trace by *Sled* , if the application has any failed speculation, it will branch to fix-up code. The execution of that fix-up code will get recorded in the trace. So modeling the ALAT is not necessary unless one is experimenting with the design of the ALAT. In that case, however, one must use *Peppermint* in execution-driven mode so that the branch to fix-up code is only taken when the modeled ALAT indicates failure of speculation.

2.1.4 Control Speculation

Control speculation allows the compiler to minimize the stall due to load instructions that suffer large latencies resulting from cache misses. The compiler can speculatively schedule such loads far ahead of their normal position in the instruction stream, even before intervening branch instructions. This indicates the load should be executed conditionally depending on the outcome of the branches. For such speculative loads, exceptions such as page faults are deferred until the outcome of the speculated branches are known. *Peppermint* currently does not model exceptions. All loads are sent to the memory hierarchy. Like data speculation, it can be added to the model. However *Peppermint* must be used in execution-driven mode, as explained in Section 2.1.3.

2.1.5 Register Remapping

In the IA-64 architecture, instructions dispersed to functional units, must undergo register remapping before being able to access the register file. There are 2 features of the IA-64 Instruction Set Architecture that require remapping of register names: register stacking and register rotation. Register stacking ensures that each active frame on the procedure call stack of a program can use its own register frame consisting of potentially all physical registers in the general register file. Hardware transparently manages the register file in two ways. It remaps the register number in the instruction to the correct physical register. It also spills and restores registers between the general register file and backing store in memory. Register rotation enhances instruction-level parallelism; it allows overlapped execution of loop iterations, known as software pipelining. This is done without requiring the compiler to unroll the loop body when generating code. This is possible since the hardware manages counters called rotating register base (RRB) that can remap the register number in an instruction to different physical registers in different iterations.

Register remapping is handled in both *Peppermint* and *Sled*. Firstly, *Sled* reads the *Current Frame Marker* (CFM) register to extract the parameters governing stacking and rotation. Then it does the appropriate modulo arithmetic to identify the physical register to be read. Since the register is spilled to the backing store during trace generation, the correct location in memory is then read. Similarly, *Peppermint* models the *Current Frame Marker* and related registers, and simulates the effect of all instructions that affect them. Examples of such instructions are *alloc*, *br.call*, *brl.call*, *br.ret*, *cover* and *clrrrb* . Furthermore, it also implements the remapping function. However it diverges from the hardware in assuming a very large physical register set. So memory traffic resulting from spilling or restoring registers is not simulated.

2.1.6 Cache Hierarchy

Peppermint simulates all three levels of cache present in the *Itanium* processor family. The simulator is highly parameterized, and the cache configuration can be easily modified to explore the design space with simulations. Parameters that could be changed for each cache include the size, line

size, associativity, line replacement policy, number of ports, access latency and number of outstanding misses. Also caches can be turned off selectively to alter the depth of the cache hierarchy. Instantiating an additional level of cache and adding it to the configuration file is quite simple. Currently *Peppermint* makes some simplifying assumptions regarding access latency. Exceptional cases are not treated separately. Instead all integer loads have a latency of 2, 6 or 21 cycles depending on whether they hit in the *Itanium*'s L1D, L2 or L3 cache respectively. Instruction fetches have a latency of 1, 5 or 20 cycles depending on whether they hit in the *Itanium*'s L1I, L2 or L3 cache respectively. Floating-point loads bypass the first level of cache, and incur a latency of 9 or 24 cycles, depending on whether there is a hit in L2 or L3 cache. All types of loads that miss in the L3 cache access memory by initiating a transaction on the system bus. The details are described in the following sections. *Peppermint* also supports a limit on the number of outstanding misses for each level of the cache hierarchy.

Changing the values for the different parameters simply requires modification of these values in the configuration file. However treating special cases separately, that are not already handled in *Peppermint*, will require addition of code to dynamically override the latency when the exception occurs. Since *Peppermint* decodes each instruction, the framework is present for such code to be added easily.

2.2 System Bus

We simulate a pipelined, split-transaction system bus. The frequency is parameterized. Arbitration is done in round-robin order among the requesting caches. The arbitration algorithm can be called by higher-priority I/O devices. However the current model does not include such devices.

2.3 Memory Controller

We have parameterized the memory controller so that the bits from the physical address to be used for row address, column address and bank can be specified in the configuration file. This allows us to study different memory configurations. Other parameters that can be varied include the number of open pages and size of outstanding read and write queues in the memory controller. Scheduling policies for memory accesses can also be studied. We schedule reads with a higher priority than writes, with some exceptions such as the presence of overlapping writes. However a whole range of policies can be studied by writing the appropriate function for selecting among the different queues for reads, writes, precharge and activation. Additional queues can also be defined if needed. However since we do not model the internal logic of the memory controller, we assume a constant time overhead in addition to the waiting time in the queues. This overhead is parameterized, and memory benchmarks could possibly be used to approximate the value for a computer system. For DRAM, important parameters available for DRAM data sheets are parameterized. Various situations such as precharge and hit on an open page are modeled. Accordingly, the latency seen by each memory access varies.

2.4 Performance and Validation

The cache system of *Peppermint* has been validated against *Dinero* [8]. Preliminary validation has indicated that the margin of error is within 20%. Further detailed validation is planned using the *McKinley* processor. The throughput of *Peppermint* will depend on the details being simulated. Table 1 gives an idea of how the throughput of *Peppermint* varies with the benchmark used and with the length of the trace. Unlike [11], we do not use sampling. All instructions are simulated. Hence our throughput numbers cannot be directly compared to theirs.

Program	Instructions	Throughput (instr / sec)
MCF	100 K	7692
MCF	1 M	11494
VORTEX	100 K	20000
VORTEX	1 M	22727

Table 1. Throughput of Peppermint for SPEC 2000 benchmark programs, MCF and VORTEX, on a HP Kayak XU 800 workstation with 733 MHz Pentium III processor and 256 MBytes RDRAM

3 Trace Generation Tool: *Sled*

Peppermint currently does only trace-driven performance simulation. A trace generation tool is required to produce the input for the simulator. Trace collection methods have been classified into at least five different types: probe-based, microcode modification, instruction set-emulation, static code annotation, and single-step execution [12]. A trace generation tool called *Sled* was developed to collect the input traces for the *Peppermint* simulator. *Sled* can single-step application processes and thus collect instruction and address traces. These traces are unfiltered since they are collected before going through the various cache levels. This feature is useful for exploring design parameters in the memory hierarchy such as the effect of changing the cache sizes, latencies or associativity.

3.1 Implementation Details

Sled currently produces traces in a modified Dinero format [8]. The basic record type is explained in [2]. *Sled* is basically a program that can control the execution of another process. Under Linux the required functionality is provided through the *ptrace(2)* system call. In HP-UX 11i the *ttrace(2)* call provides similar function. These system calls provide support to manipulate the execution state of a process. They can be used to single-step the execution, to read and modify most of the registers of a stopped process, as well as to read any location of the memory space of the process. The register reading facility is used extensively by *Sled* to collect instruction trace data by reading the Program Counter. Memory locations are read to extract the current execution bundle from the process image.

A monitored process is usually created by *Sled* forking and then executing the program of interest. However, *Sled* can also be attached to a running process by specifying the process id. If *Sled* forks a child, the new process puts itself into a traceable state and waits for the parent. The parent *Sled* process can set a breakpoint if specified. It then signals the child to proceed. The child process executes the program of interest, which runs till the breakpoint. The hardware debug registers of the *Itanium* are used to set these breakpoints, but a similar objective could be achieved by modifying the programs code space and inserting a software breakpoint.

Once the child reaches the first breakpoint, the parent can collect statistics, and disable this breakpoint. Trace collection would typically be enabled only after reaching the first breakpoint. If the last breakpoint is specified, it is set at this time. This allows *Sled* to collect data even within a 'backward' window in the program.

Depending on the arguments to *Sled* , the child program either proceeds or is single-stepped till it exits or reaches the last breakpoint. At this point, the parent can stop generating trace records, or get the current values of the PMU counters. The parent process detaches itself from the child. The child process can then run to completion. As a convenience, a termination signal can also be sent to the child at this time. This is especially convenient for collecting data in a limited range of a long running program.

In tracing mode, *Sled* reads the value of the instruction pointer from the context of the child at every trap point. The current instruction bundle is then read by dereferencing this value and reading from the memory image of the process being monitored. Memory accesses are then detected by disassembling the contents of the bundle. The bundles are decoded using code from the IA-64 version of the *GNU binutils*. If a memory access is found, its address is read from the corresponding indirection register. In case of *Itanium*, this may require reading registers from the backing store. If a memory access occurs in a loop, the registers also have to be de-rotated before they can be read. *Sled* also determines the contents of the predicate register and thus creates the appropriate record entry. The decoded bundles are cached in a direct mapped buffer. This saves the cost of system calls for the next time that these instruction bundles would have to be read from the memory image of the process.

In addition to collecting address and instruction traces, *Sled* can also be used as a performance measurement tool. It uses the *perfmonctl* system call available in the *Linux IA-64* kernel to access the performance monitoring registers available on the *Itanium*. *Sled* provides the capability to collect detailed statistics around any window of execution of a program. One extension planned for *Sled* is the ability to reset breakpoints, so that one can accumulate the performance data for a region of code that get executed repeatedly. This overcomes one fundamental limitation in the *Itanium* implementation of the Performance Monitoring Unit. Specifically, address range based performance monitoring is not available for all possible event types, for example CPU_CYCLES [6].

3.2 *Sled* Performance

One metric proposed for evaluating the quality of a trace collection tool is the slowdown compared to a full speed execution of the same piece of code [12]. The slowdown reported for single-step based trace collection cover a wide range, from 100 [1], 1000 [4], to 10,000 [5]. As shown in Table 2, the slowdown as seen on a 800 MHz dual *Itanium* system lies towards the low end of reported figures. p

Program	Instr	Full Speed	Trace Enabled	Slowdown
	(Millions)	(sec)	(sec)	
MCF	3.17	1.08	59.23	60
VORTEX	1.26	0.21	41.7	200

Table 2. Trace collection slowdown for SPEC 2000 benchmark programs, MCF and VORTEX

Trace collection is more expensive than merely single-stepping the processor, since system calls are required to read the instruction pointer, the bundles, and in case of a memory access, the contents of the indirection register, the predicate register and the backing store. However, the bundle cache performs fairly well as seen by the data in Table 3. It is can be seen that the single stepping cost and trace collection cost both scale linearly with the length of the trace

collected, and that the overhead for trace collection lies between 20 - 30 % of the execution time.

Instruction Count	Single step (sec)	Trace collect (sec)
1 Million	23.49	28.44
10 Million	211.14	270.38
100 Million	2174.00	2691.59

Table 3. Single stepping and Trace collection time for SPEC 2000 benchmark program, MCF

4 Experimental Evaluation of Systems

Next we present results based on experimental evaluation of systems based on *Itanium* and *McKinley* processors. Systems based on *Itanium* and *McKinley* processors differ not only in the processor, but also in the system bus and memory. *Peppermint* allows us to explore the design space by doing controlled experiments. We compare 2 systems based on these 2 processors, with typical configurations for system bus and memory, to get an estimate of the performance improvement for different applications between these 2 systems. We also vary design parameters to get a sense of what contributes to the difference in performance. The differences between the systems compared are presented in Table 4. More detailed comparison can be made by comparing the configuration files for *Itanium* and *McKinley* presented in the Appendix of [2].

Parameter	*Itanium*	*McKinley*
Processor Frequency	800 MHz	900 MHz
Third-level Cache	4 MB	3 MB
System-bus Frequency	133 MHz	200 MHz
System-bus Width	64 bits	128 bits
Double Data Rate DRAM	No	Yes

Table 4. Some Difference in system parameters between the 2 systems

5 Results

In the first set of experiments shown in Figure 2, we compare the execution of *Itanium* and *McKinley* based systems compared in Table 4. The metrics used are instructions per cycle (IPC) and instructions per second (IPS). Each line shows the corresponding metric for *McKinley* normalized to that of *Itanium*. The results show that for the application traces used, the performance improvement of *McKinley* over *Itanium* can range from 7% to over 100%, when comparing IPC. With IPS, the results are even better, since the frequency of the *McKinley* processor used in our experiments is higher than that of *Itanium*.

In the second set of experiments shown in Figure 3, we take the total execution cycles of each application, normalized to the longest running application, and plot a bar graph. Each bar shows 5 components. The *Busy* component counts cycles in which at least 1 instruction is dispersed by the front end of the processor pipeline. The remaining components measure stalls in instruction dispersal. They are attributed to instruction fetch (*Ifetch*) when the stall results from the decoupling buffer being empty. If the stall occurs because an issue port cannot be assigned to the instruction, it is attributed to functional units (*Funit*). If a data dependency forces instruction dispersal to be stalled, it is accounted for in the *DataDep* component. Finally the *Cache* component account for stalls reached when loads or stores cannot be issued due to the limit on outstanding misses in the caches.

On the X-axis, each bar has a label. The first character indicates the application, which can be *Gap (G)*, *Transcoder (T)*, *Mcf (M)* or *Vortex (V)*. The second character indicates whether the system simulated is an *Uniprocessor (u)* or *Multiprocessor (m)*, specifically a 2-processor SMP. The last character indicates *Itanium (I)* or *McKinley (M)*, indicating the system parameters used, as shown in Table 4.

Both *gap* and *transcoder* are quite compute-intensive and most of their execution time can be attributed to *Busy* cycles in which 1 or more instructions are issued. The *FUstall* component of *transcoder*, which represents stall due to unavailability of functional units, decreases from 13% for *Itanium* to 8% for *McKinley*. This is due to the increased number of functional units in *Mckinley*. This increases the IPC of *transcoder* by 7%. For *gap*, the improvement in execution time from *Itanium* to *Mckinley* can be attributed to both *DDstall* and *IFstall*, stalls due to data dependencies and instruction fetches respectively. The larger size and line size of the second-level cache in *Mckinley* benefits *gap*. It also benefits significantly from the better memory system of *Mckinley* (double-data rate and higher frequency), which implies lower latency for third-level cache misses. Its IPC increases by 11%. For the dual-processor configurations, both applications maintain their performance improvement from *Itanium* to *Mckinley*.

The uniprocessor execution of *mcf* on *Itanium* and *Mckinley* show a huge difference in performance. This is because the execution of *mcf* on *Itanium* is dominated by data dependency stalls (*DDstall*). It accounts for 86% of the execution time. *Mckinley* cuts down this component to 75%. This improves the IPC of *mcf* by 100%. This is despite the

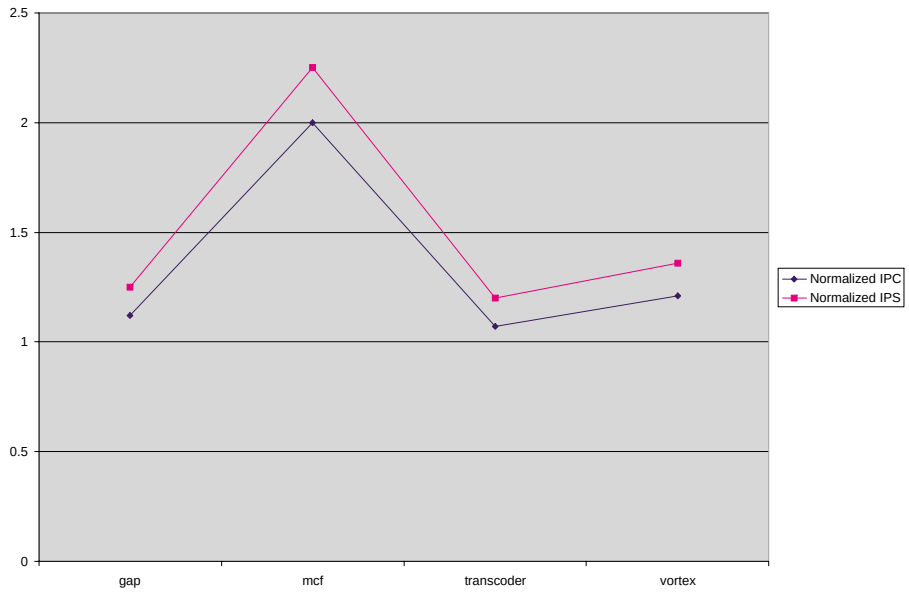

Figure 2. Relative IPC and IPS

fact that *Mckinley* has 3 MBytes of third-level cache, compared to 4 MBytes in *Itanium*. The primary contributor for this performance improvement is the better memory system of *Mckinley* as explained in the previous paragraph. The memory system is critical for *mcf*, because *mcf* is pointer-intensive, and has low cache hit rates. The problem is more acute in IA-64 architectures, where the pointers increase to 64 bits in width, and result in larger working sets. For the dual-processor execution of *mcf* with *vortex*, we find that improvement in IPC is only 36% from *Itanium* to *Mckinley*. This is due to the higher contention for system bus and memory in the dual-processor systems. For *mcf* on the uniprocessor systems, bus utilization was 71% and 76% for *Itanium* and *Mckinley* respectively. For the dual-processor systems, we found the bus utilization to be 77% and 79% for *Itanium* and *Mckinley* respectively. The number of memory accesses that had to be retried because the memory controller had reached the limit on pending reads or writes was 5.2 million and 2.7 million on *Itanium* and *Mckinley* respectively in the uniprocessor case. For the dual-processor case,the number of retries were 7.3 million and 3.0 million on *Itanium* and *Mckinley* respectively. The average latency of a read serviced by memory was 103 processor cycles for *Itanium* and only 72 cycles for *Mckinley*; difference between uniprocessor and dual-processor configurations was negligible. This implies that memory bank conflicts did not change significantly with increase from 1 to 2 processors. To summarize, the main problems observed in the SMP case are high bus utilization and increased retries as a result of queues in the memory controller reaching their limit. So improving system bus bandwidth and increasing the queue sizes in the memory controller might be the first steps in improving SMP performance.

The uniprocessor runs of *vortex* on *Mckinley* yields an improvement of 21% in IPC over *Itanium*. This is a result of reduction in stalls due to data dependencies (*DDstall*) and instruction fetches (*IFstall*). In the dual-processor case, the improvement in IPC for *vortex* is only 16%. Again this can be attributed to the higher contention for system bus and memory. The number of retries induced by the memory controller in the dual-processor case, as mentioned earlier, were 7.3 million and 3.0 million on *Itanium* and *Mckinley* respectively. For the uniprocessor case, *vortex* had 1.8 million and 0.8 million retries for *Itanium* and *Mckinley* respectively.

5.1 Parameter Sensitivity

We also studied variation of some system parameters. Figure 4 shows the effect of varying the frequency of the system bus. The total execution cycles of each application is plotted relative to the longest running application, which is normalized to 100. Each bar is labeled on the X-axis with 1 or 2 applications (G, M, T or V), followed by the system architecture (I or M) and bus frequency. For the *Itanium* system configuration, the bus frequencies chosen are 133, 266 and 333 MHz, corresponding to labels 1, 2, and 3 respectively. For *McKinley*, the corresponding bus frequencies are 200, 333 and 450 MHz respectively. The most significant improvement in performance has been observed

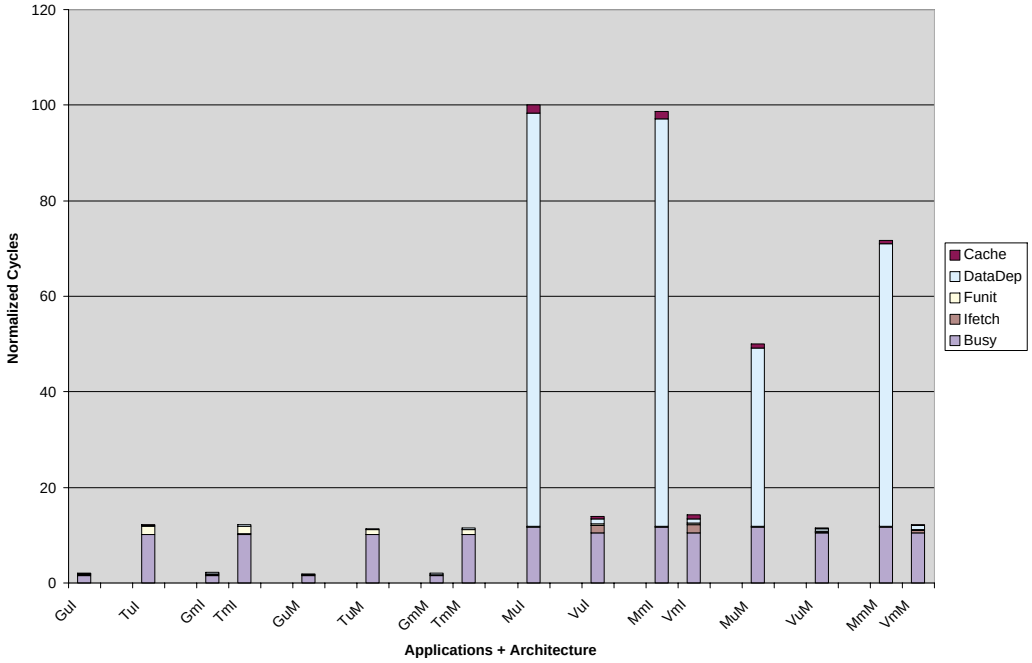

Figure 3. Breakdown of normalized execution cycles

in *mcf*. The *DataDep* component shows significant reduction for this application on both *Itanium* and *McKinley*. On *Itanium*, for example, its IPC increases by 42% if the bus frequency is increased from 133 to 333 MHz.

Figure 5 shows the effect of varying the size of the third-level cache. The total execution cycles of each application is plotted relative to the longest running application, which is normalized to 100. Each bar is labeled on the X-axis with 1 or 2 applications (G, M, T or V), followed by the system architecture (I or M) and finally the size of the third-level cache (3, 4, 8 or 16 M). Increasing cache size yields better performance improvement compared to increasing bus frequency. In fact, the IPC for *mcf* on *Itanium* increased by 42% due to increase in system bus frequency, while it increased by 120% due to increase in cache size. The results turned out to be additive. That is, the 2 enhancements together increased IPC by 162%. For *McKinley*, the improvement in IPC resulting from increasing bus frequency to 333 MHZ was 30%, while increasing third-level cache size to 8 MBytes increased IPC by 113%. Together, these 2 enhancements increased IPC by 126%.

6 Conclusion

This paper describes the implementation of *Peppermint* and *Sled*: tools for cycle-accurate, trace-driven simulation of applications running on IA-64 processors. We find that the improvement in IPC of *McKinley* relative to *Itanium* ranges from 7% to over 100% for our different experiments. The improvement can be attributed to a variety of factors. These range from the availability of additional functional units and issue ports in the *McKinley* processor to our assumption of a better memory system. While the improvement in performance remains valid in SMP systems in some cases, higher contention for system bus and memory reduces the performance gain in other cases. Increasing the system bus bandwidth and size of queues for pending requests in the memory controller are identified as first steps for optimizing SMP performance.

References

[1] A. Agarwal, J. Hennessy, and M. Horowitz. Cache Performance of Operating System and Multiprogrammed Workloads. *ACM Transactions on Computer Science*, 6(4):393 – 431, 1988.

[2] S. Basu, S. Roy, R. Kumar, T. Fisher, and B. E. Blaho. Peppermint and Sled: Tools for Evaluating SMP Systems based on IA-64 (IPF) Processors. Technical report, Hewlett-Packard Labs, 2002. Available from http://www.hpl.hp.com/techreports/index.html.

[3] D. Burger and T. Austin. The SimpleScalar Tool Set. Technical Report 1342, University of Wisconsin, 1997.

[4] J. K. Flanagan, B. E. Nelson, J. K. Archibald, and K. Grimsrud. BACH: BYU Address Collection Hardware, The Col-

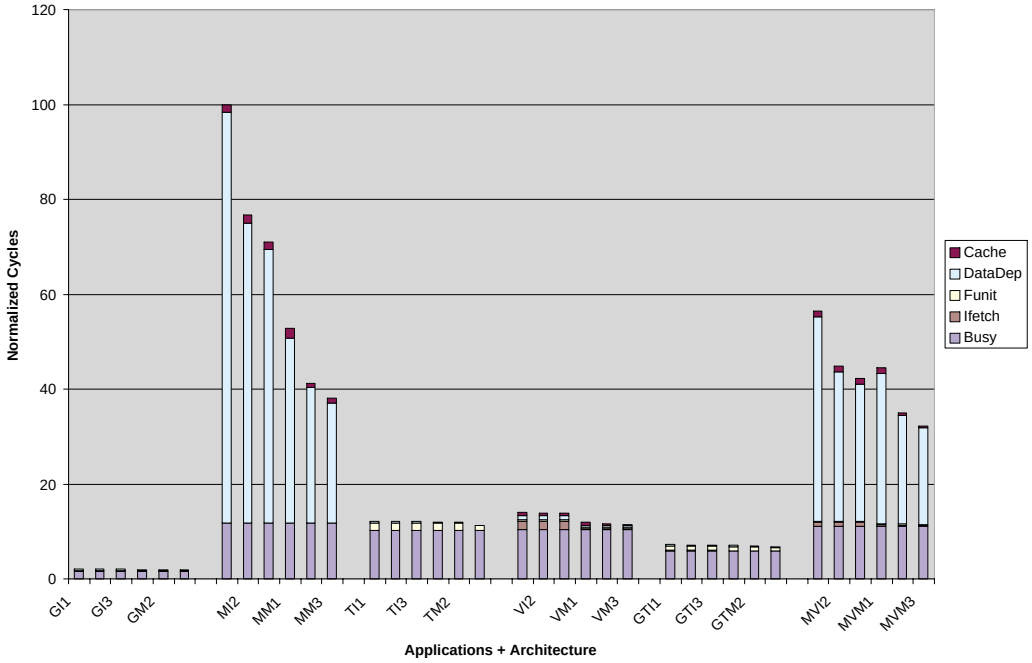

Figure 4. Effect of Bus Frequency on Execution Time

lection of Complete Traces. In *Proceedings of the 6th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 128 – 137, 1992.

[5] M. Holliday. Techniques for Cache and Memory Simulation using Address Reference Traces. *International Journal in Computer Simulation*, 1:129 – 151, 1991.

[6] intel. *Intel®IA-64 Architecture Software Developer's Manual*, volume 4, chapter Processor Performance Monitoring. Intel Corporation, 1.1 edition, July 2000.

[7] Intel Corporation. *Itanium Processor Microarchitecture Reference*, August 2000. Document Number 245473-002, Available from http://developer.intel.com/design/itanium/manuals/.

[8] Jan Edler and Mark D. Hill. Dinero IV Trace-Driven Uniprocessor Cache Simulator. Technical report, University of Wisconsin, 1999. Available at http://www.cs.wisc.edu/ markhill/DineroIV/.

[9] A.-T. Nguyen, M. Michael, A. Sharma, and J. Torrellas. The augmint multiprocessor simulation toolkit for intel x86 architectures. In *International Conference on Computer Design (ICCD)*, 1996.

[10] H. Sharangpani and K. Arora. Itanium processor microarchitecture. *IEEE Micro*, pages 24–43, September-October 2000.

[11] R. Uhlig, R. Fishtein, O. Gershon, I. Hirsh, and H. Wang. Softsdv: A presilicon software development environment for the ia-64 architecture. *Intel Technology Journal*, 4th Quarter 1999. Available from http://developer.intel.com/technology/itj/.

[12] R. A. Uhlig and T. N. Mudge. Trace-driven Memory Simulation: a Survey. *ACM Computing Surveys*, 29(2):128 – 170, June 1997.

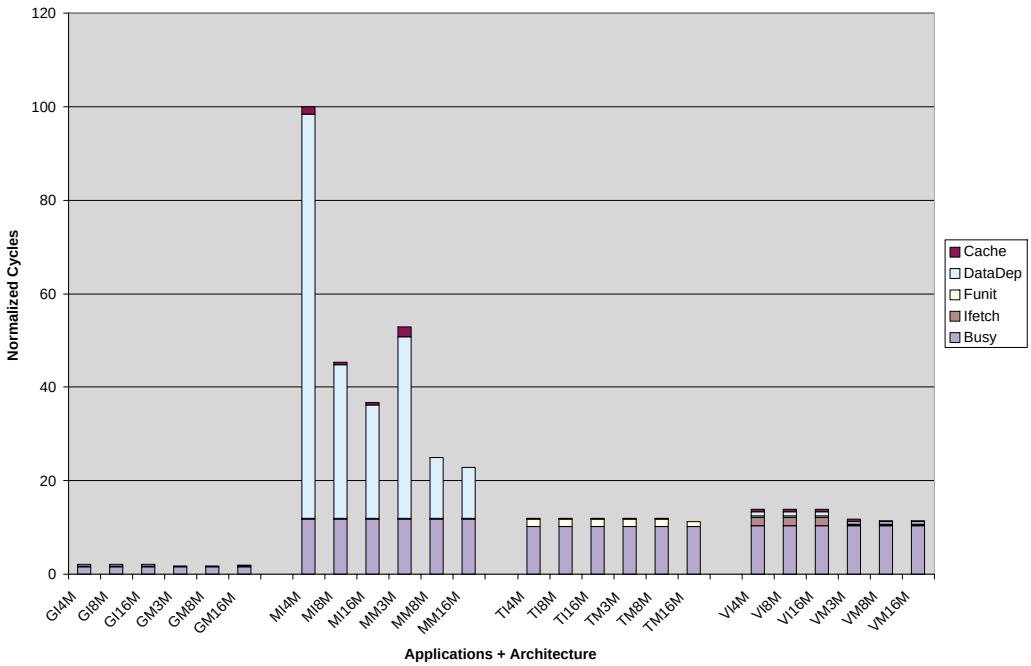

Figure 5. Effect of Level 3 Cache Size on Execution Time