Using Software-Extended Architectures for Software Simultaneous Multithreading

Emmett Witchel and M. Frans Kaashoek

MIT Laboratory for Computer Science
545 Technology Square
Cambridge MA 02174

Email: {witchel, kaashoek}@lcs.mit.edu

Janurary 27, 1997

Abstract

A *software-extended architecture (SEA)* enhances a hardware architecture by placing a high-performance dynamic instruction-set translator between the application binary and the processor, improving processor utilization and enabling new functionality with no changes to either the processor or the binaries. Our prototype implementation of a software-extended Alpha 21164 can provide new system functionality while adding only 1%–30% to the running time of an application. Using this prototype, we have implemented *software simultaneous multithreading (SSMT)*, a new software technique for allowing programs to make greater use of the processor pipeline. SSMT merges instruction streams from independent processes, in order to increase instruction-level parallelism. Experiments with SSMT on the software-extended Alpha 21164 show that processor throughput can be improved by up to 30% on real programs, despite the small number of issue slots on this processor.

1 Introduction

While innovations in computer architecture like multiple issue slots, dynamic scheduling, and large register sets are becoming commonplace, limitations in compiler technology and inherent data-flow requirements prevent these capabilities from being fullly used. Furthermore, successive implementations of the same architecture often have different performance characteristics, penalizing legacy binaries. In addition, some users would like to add new capabilities to their architecture, e.g. system call redirection, without having to modify their hardware.

Software-extended architectures (SEA) are a novel approach that addresses these issues. A software-extended architecture interposes a dynamic instruction-set translator between the application binary and the processor. At runtime, application code is read by the translator, and new functionality is added; the result is immediately executed. The translator does not require modifications to the executable, so existing binaries can be augmented with new capabilities. System call redirection [Jon93], sandboxing [WLAG93], preemptive user-level thread scheduling, and using configurable processing hardware [RS94] are natural applications for software-extended architectures. Software-extended architectures can also dynamically perform chip-specific optimizations, such as instruction scheduling, and data-dependant optimizations based on feedback information. The key advantage of software-extended architectures is that users benefit without changing the processor or the application binary.

The translator for the SEA prototype described in this paper is based on recent advances in dynamic binary translation for fast machine simulation [CK94, WR96] . The primary contributions of our translator implementation are removing binary translation from the machine simulation context and adding additional optimizations (such as code scheduling and dynamic inlining) that keep translator overheads low and emitted code quality high. The result is a dynamic translator that can extend the capabilities of existing architectures and improve performance for real applications.

To illustrate the value of software-extended architectures we added support for *software simultaneous multithreading (SSMT)* to the Alpha 21164. With the pipelines of superscalar processors becoming deeper and wider, techniques for improving pipeline utilization are becoming more important. Like simultaneous multithreading (SMT) [TEL95], SSMT increases application throughput by increasing pipeline utilization. Unlike SMT, SSMT gains this performance without radically changing the underlying machine architecture. The SSMT runtime system dynamically merges instruction streams from different processes, thereby increasing instruction-level parallelism and making that parallelism easier to exploit. By proper code scheduling, the latency of one process's register dependencies can overlap either instruction execution or register dependence latency of another process. SSMT increases system throughput at the cost of some per-process latency.

The work in this paper does not address the interface to the SSMT system. Nevertheless, one of the more intriguing possibilities would be for the operating system scheduler to choose one or more kernel threads from the run queue for merging. Thus, the only program whose text is directly executed on the processor is the operating system itself. This option has profound architectural implications as it frees processor architects to make drastic changes for every chip revision—the runtime system will ensure binary compatibility for user programs. In addition, the runtime system can greatly simplify processor design by performing any necessary register renaming, dependency checking, and code scheduling.

There are a variety of static translation tools (see Section 2) which share some of the goals of SEA. While some SEA functionality could be implemented statically, other applications, such as SSMT, require dynamic translation. If SSMT were implemented using static translation, each possible execution path of one program must be merged with every possible path of the other program; furthermore, this process must be repeated for all application pairs which might run in tandem. The storage overhead of such a scheme makes it unrealistic. Run-time translation enables data-dependent optimization, such as those exploited by dynamic code generation systems [Eng96], as well as SMT. It is possible that static translation is sometimes more appropriate, but both techniques are valuable, and they can often be used in conjunction.

The key contributions of this paper are the design and implementation of SEA and SSMT. We show that for our prototype implementations on the Alpha 21164, an SEA can provide services while adding only 1%–30% to program execution time of some benchmarks, and SSMT can increase processor throughput by as much as 30% for some application pairs. While performance gains are possible, the 21164 is not a target architecture for SSMT. The performance of our prototypes are limited by the narrow issue width of the 21164. Our measurements indicate that for processors which have more registers and/or wider pipelines, SSMT should perform substantially better. In the long term, software-extended architectures (and SSMT) could allow a VLIW-like processor to run standard RISC binaries, or allow superscalar architects to simplify their design and concentrate on increasing clock rate and adding more functional units.

The rest of this paper is organized as follows. Section 2 presents related work in software-extended architectures, simultaneous multithreading and binary translation systems. Section 3 presents the control flow and main data structures of the dynamic binary translator and the new optimizations it implements. It also presents our experimental environment and a quantitative analysis of the cost of a software-extended Alpha 21164. Section 4 discusses how a software-extended architecture can be used to implement SSMT. It explains our merging algorithms and evaluation methodology, and provides a quantative analysis the gains attainable from SSMT on the 21164. Section 5 discusses some of the architectural features that would benefit the translation system and discusses the implications of SEA.

2 Related work

In this section we discuss recent related work in hardware simultaneous multithreading and software binary translation. The hardware multithreading project that SSMT most closely resembles is simultaneous multithreading (SMT) [TEL95, TEE$^+$96]. A SMT processor dynamically partitions chip resources for multiple hardware threads, gaining an advantage over the static partitioning done by multiprocessors. Multiple hardware threads, increase the instruction level parallelism available to the processor.

The goals and benefits of SSMT are similar to SMT, but SSMT supports multithreading in software. The advantages of a software approach is that it is able to directly leverage current trends in processor design. Instead of committing to a radically different chip design, processor architects designing for SSMT can extrapolate techniques that are currently bearing fruit.

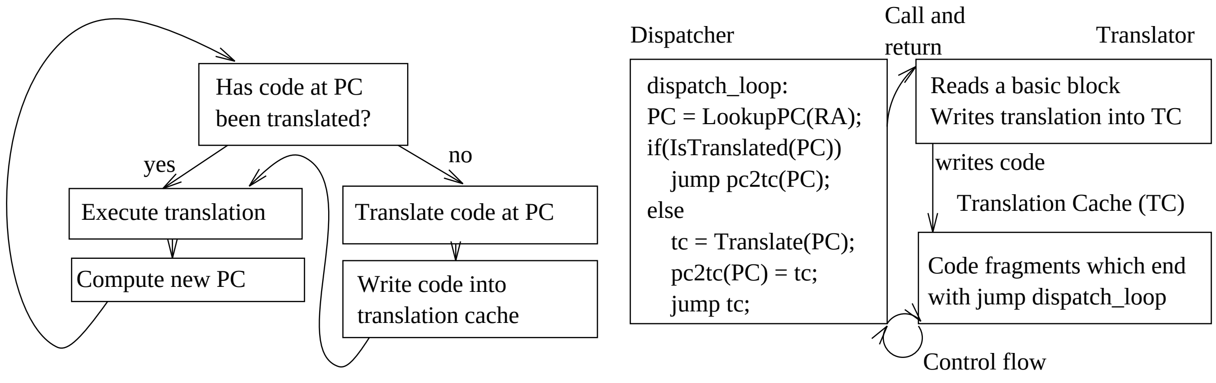
A disadvantage of using a SMT processor is that it exposes multithreading to the operating system, necessitating a kernel with locks and synchronization overhead. While multiprocessor operating systems have such capabilities, they are generally inferior in performance to uniprocessor operating systems [RBH$^+$95]. SSMT does not require a multiprocessor operating system.

Another architectural drawback of the SMT design is its need for a large, coherent register file (8*32+100 renaming registers in the simulated architecture). While SMT uses two processor cycles to access this large register file [TEE$^+$96], this design could limit the clock rate of the chip. SSMT does not need a single large register file because register renaming is done in software.

An advantage of the SMT approach that it is likely to scale to a larger number of threads than SSMT, as the exponential growth of possibilities for conditional branch resolution quickly becomes intractable with SSMT, even with architectural support.

Software-extended architectures are inspired by ideas and implementation techniques used in binary rewriting tools. Earl Killian's Pixie [Smi91], ATOM [SE94], EEL [LS95] and similar tools allow users to analyze and instrument whole program binary images. Image modification allows extensive analysis and optimization, since the work is done off-line and can be amortized over many executions.

Some systems [SCK$^+$93] use a hybrid approach in which most work is done statically, but run-time support is provided for difficult cases like self-modifying code. Just in time compilers [Gos95] and virtual execution environments [ATLLW96] perform static translation (including chip-specific optimizations), but the user must wait for the translation to be complete before execution starts. Dynamic translators such as Shade [CK94] and Embra [WR96] translate binaries incrementally at run-time. The run-time system for software extended architectures borrows implementation techniques from these fully dynamic systems.


Has code at PC been translated?
yes
no
Execute translation
Compute new PC
Translate code at PC
Write code into translation cache
Dispatcher
Call and return
Translator
Reads a basic block Writes translation into TC
writes code
Translation Cache (TC)
Code fragments which end with jump dispatch_loop
Control flow

```
dispatch_loop:
PC = LookupPC(RA);
if(IsTranslated(PC))
   jump pc2tc(PC);
else
   tc = Translate(PC);
   pc2tc(PC) = tc;
   jump tc;
```

(a) Control flow (b) Software architecture

Figure 1: Control flow and software architecture for the dynamic program translator. If the code at the current program counter is translated, then we execute a cached copy of the translation. Otherwise, we translate the code, write it into the translation cache, and then execute it.

Some recent concurrent efforts are using translation technology, binary rewriting tools, and runtime-feedback to improve performance of binary applications. A recent product, FX!32 [Dig], translates codes from the x86 to the Alpha, and uses runtime feedback to improve the quality of the translated code. Our SEA implementation also uses runtime feedback, and it could profit from the runtime statistics databases maintained by FX!32. Morph is a new project that plans to use profile-based information to re-optimize executables [CSB96]. SEA shares some performance goals with these static translation schemes. We feel that dynamic translation has some advantages in allowing binaries to be optimized for specific classes of inputs, and by enabling data-dependent optimizations. But we also believe that our approach is compatible with static tools. For example, one might use FX!32 to statically translate from the x86 instruction set to the Alpha instruction set and then run the resulting binary on a SEA.

Finally, we note that sofware-extended architectures represents an application of RISC [Pat85] philosophy. While RISC pushed hardware complexity into software, an SEA allows new hardware functionality to be implemented in software.

3 Software-extended architectures

The key component of a software-extended architecture is the dynamic instruction translator. In this section, we describe the basic design of the translator and some of the optimizations that it employs to make software-extended architectures practical. We also discuss the proper division of labor between a compiler and a SEA, and evaluate the performance of our SEA prototype implementation.

3.1 The dynamic translator

The control flow of the the dynamic program translator is conceptually similar to fast machine simulators such as Shade [CK94] and Embra [WR96] (depicted in Fig. 1a). The translator reads a program binary finds its entry point, and begins translation. The *fetch unit* of the translator reads code from the application text segment, performs various functions on the code like register renaming, dependency checking, and pipeline scheduling, and then writes the result to the *translation cache*, an area of memory which holds translations. The work done by the translator is amortized because a cached translations are executed repeatedly. This translation process is analogous to the predecode phase of some modern processors, where code is read from memory, and then written in a slightly modified form to the instruction cache.

Figure 1b illustrates the software architecture of these dynamic translation systems. The main dispatch loop determines whether the translator needs to be invoked. It does this by using the current program counter as an index into the *pc2tc* hash table, which returns the location of the translated code for this program counter, if a translation exists. If the translation does not exist, the translator is called, it writes the translation into the translation cache, and the location of the translation is recorded in the *pc2tc* hash table.

3.1.1 Performance of the dynamic translator

The performance overhead of the SEA runtime (i.e., the translator and the dispatch loop) is due to two sources, (1) extra instructions in the code translations and (2) support routines like the translator and the dispatcher. The SEA runtime adopts the litany of optimizations implemented in Shade and Embra. These optimizations greatly reduce the amount of time spend in the translator and dispatcher so that almost all execution time is spent executing translations. Since a SEA translator must be more concerned with the quality of its generated code than either Shade or Embra, it contains new optimizations to address the overhead of extra instructions. These optimizations are discussed below in Section 3.2.

The SEA runtime also adds architectural overheads to program execution time in the form of increased cache and TLB miss rates, but the measured impact of these overheads is small (see Section 3.4).

3.2 Optimizations performed by the SEA runtime

The SEA runtime adds instructions to the workload to perform functions like tracking the current PC. The performance impact of these instructions can be reduced by (1) minimizing the number of added instructions, and mitigated by (2) scheduling the instructions intelligently and (3) spreading the cost of the bookkeeping over larger translation units. In Section 3.2.1, we enumerate the tasks performed by the SEA runtime in its translations, and how it can perform those functions efficiently. We also talk about how the SEA runtime can try to hide these costs.

Adjusting the fetch policy of the translator has potential advantages and disadvantages that are algorithmic and architectural. An algorithmic advantage of following branches is that the translator can fetch larger translation units, which can be better scheduled, and which can reduce the amount of necessary bookkeeping.

Some form of branch prediction is necessary to fetch through conditional branches (which are much more prevelant than unconditional branches). Information to increase prediction accuracy can be collected at run-time, or read from feedback information.

Because branch prediction is never perfect, the algorithmic disadvantage of an aggressive fetch policy is that the translator is doing useless work by fetching down a path which is not executed. Additionally, translations that waste space in the translation cache can cause the cache to fill (which is expensive to deal with) more quickly.

Architecturally, fetching through branches has the potential to increase the cache and TLB locality of translations because if the branch prediction is accurate, code for a long path of instructions may reside in a single translation unit. However, if only small parts of each large translation unit are executed (due to poor branch prediction), control might be transfered all around the translation cache, reducing cache and TLB locality.

Since many basic blocks might transfer control to a single block, following branches can also lead to code replication. Code replication can effectively trade increased space for reduced time, but it can also put pressure on the underlying memory hierarchy.

The fetch policy of the translator is a key parameter of the translator, and is analogous to the hardware branch-speculation policy of modern processors. Because the translator is implemented in software, its policies can be aggressively tailored for a given application. While the literature on hardware branch-prediction is large [YP92, MEP96, GYCS96], we believe that SEA could open up a new trade-off space for branch prediction algorithms based on information collected by SEA software. These new branch prediction schemes could either replace hardware prediction, or work with hardware prediction schemes.

Like other SEA functions, software collection of branch information adds instructions to code translations. While algorithmic innovations for instruction scheduling can help address this cost, the key technological trend that SEAs anticipate is wider issue widths. Current pipelines are too narrow to hide the overhead instructions of an SEA (as measured in Section 3.4), but we expect that technological trends will address this problem.

3.2.1 SEA runtime bookkeeping

The SEA runtime adds instructions to its translations to track the current program PC, implement register indirect jumps and, support register reallocation.

The SEA runtime needs registers for its bookkeeping tasks. In order to minimize the need for hardware support, the runtime does not require dedicated registers, rather it spills program registers when needed. The runtime uses a static tool (i.e. the program is not executed) to analyze a binary image and produce a file detailing the binary's register usage. The runtime reads this file, and when possible, it uses "dead" registers (a register whose next appearance in program execution is as a destination), because the contents of these registers does not need to be saved and restored.

PC tracking. In order to continue program execution, the dispatcher needs to know the current program counter. Shade and Embra tracked the program counter value using a machine register. The SEA runtime aggressively minimizes its use of hardware resources, and so does not use a register. Instead, the return address of the branch used to exit the translation cache is used as an index into a table (this is depicted in the LookupPC step in Fig. 1b) which contains the new program PC. When code is translated, the addresses of its exit points are associated with the program counter at those points so the lookup always succeeds.

In the common case of exiting a block that ends with a conditional branch, the branch condition must be evaluated. Evaluating the condition, spilling a known register, and branching out of the translation cache are the instruction overheads for maintaining the program PC in this case.

If the translator can effectively fetch through branches, the PC tracking code can be reduced to a single check for conditional branches that the branch was predicted correctly.

Register-indirect jumps. When control is transfered to a register value, there is no way to know the destination statically. Embra had a special strategy for dealing with register-indirect jumps called *speculative chaining.* Each register-indirect jump transfers control to the prelude of the translation of the expected jump destination, which checks to see that the program counter was correct. This was a substantial win on the MIPS architecture since the MIPS compilers use register indirect jumps for procedure calls.

Procedure calls on the Alpha are generally implemented as a special type of unconditional branch, so most register indirect jumps in Alpha code are procedure returns. Because functions are usually called from diverse program sites [Wal86], procedure returns are far less predictable than procedure calls, and speculative chaining is no longer a performance win.

We discuss the details of how register indirect jumps are handled in Section 3.4, but they are expensive (≈15 instructions for base SEA and 26 for SSMT). Since the return point of a function is known at its call site, if a fetch policy can fetch through a function call to its return, it can simply continue fetching after the call site. This completely eliminates the indirect jump of the procedure return.

Register reallocation. Register reallocation is only an issue for SSMT. It is discussed in Section 4.6.1.

Hiding SEA overhead. Once the number of instructions required for an SEA task is minimized, the performance overhead can be minimized by instruction scheduling and by amortizing the cost over larger translation units.

Fetching through conditional branches allows large increases in translation unit sizes which provides an opportunity for the SEA code scheduler to revisit some compiler decisions regarding code scheduling, register allocation, and loop unrolling. There is an opportunity for cooperation between the compiler and the SEA runtime, allowing the compiler to focus on source-to-source and machine specific optimizations, while the SEA uses architecture implementation specific information for further optimization.

3.3 Drawing the line between the compiler and an SEA

The SEA runtime performs some of the same functions as a compiler, so it is important to consider which tool is more appropriate for which function.

The SEA code scheduling algorithm is simple, but chip-specific. Therefore, we would expect it to perform better than the generic schedule one would obtain from, for example, gcc, but worse than what one could obtain from the most sophisticated algorithms used in the DEC compiler's chip-specific scheduler.

However, it is not clear that the compiler is the appropriate tool for chip specific optimizations. Given that a single binary is often executed on multiple revisions of a chip architecture, we believe that chip-specific code scheduling is more appropriately done by a software extensible architecture than by a compiler, or a binary rewriting tool. However, even if a system used a static tool for chip-specific scheduling, and users weren't burdened with figuring out which binary to run on which platform, SEA can integrate data-dependent optimizations with its code scheduling. Instead of having a compiler or binary rewriting tool transform a program for its behavior on one of its inputs, a program running under a software extended architecture can provide a data file containing profile information for each class of inputs.

The SEA runtime is capable of doing aggressive inlining and loop unrolling. The compiler and an SEA seem directly compatible in this case. In order to support multiple compilation units, it is difficult for compilers to support inlining of library calls. While such inlining could be done by the linker [Wal86], the question of what to inline is difficult to answer in a machine-independent way. The SEA runtime is an extension of the architecture, so it is an appropriate level at which to make these decisions.

Finally, several studies have shown benefits to scheduling code across multiple basic blocks (e.g. trace scheduling [Fis93, Wal91]). This is a data-dependent code transformation[Fis81] and so seems appropriate for an SEA, where profile information can be provided at runtime, and the transformations can be aggressive without worrying about penalizing cases where the program behavior might be different.

To make a convincing case, and to separate out runtime feedback, we evaluate the performance

Benchmark	IPC		SEA Runtime
	Native	SEA	time
mandel	0.49	0.60	0.32%
swim	0.59	0.56	1.04%
alvinn	0.46	0.49	0.35%
hydro2d	0.49	0.47	1.12%
turb3d	1.05	0.97	1.61%
applu	0.70	0.63	2.08%
ijpeg	1.17	1.07	0.98%
mgrid	1.69	1.40	4.85%
compress	0.77	0.75	0.50%

Table 1: IPC and SEA runtime system overhead for several benchmarks. Overhead due to SEA support routines is low, but SEA adds extra instructions to the workload. How these instructions are scheduled influences the IPC.

of our SEA by compiling our benchmarks with chip-specific optimization, and neither the compiler nor the SEA runtime use profile information.

3.4 The measured performance of SEA

In order to understand the performance implications of a software-extended architecture, we present performance results and measurements of a software-extended Alpha 21164. These experiments measure the base performance of the SEA system. Some types of functionality (e.g. system call monitoring) could be performed at this performance level.

Our experiments were run on a 266MHz Alphastation 500/266 running Digital Unix 4.0. This machine has 8KB direct-mapped primary instruction and data caches with a 2 cycle access time, a 96KB 3-way secondary cache with an 8 cycle access time[ER], and a 2 MB third level, off-chip cache with a 15.5 cycle access time as measured by lmbench [MS96]. Each machine has 128 MB of memory and was in multiuser mode during the measurements. In fact, the machines were often in use during the measurements. To mitigate the effects of other processes, we compare the user portion of execution time. The user time measures all of the overheads of our system, except for additional system time induced. The increased system time due to increased TB misses (due to SEA support data structures) is not included. However, this increase is small relative to the running time of the benchmarks, and if SEA is implemented as part of the operating system, special support (e.g., superpages) could be used to eliminate it. Cache miss counts and pipeline information are measured using the 21164 on-chip counters.

To evaluate SEA we used several benchmarks from SPEC95 [SPE95], both floating point (*swim*, *hydro2d*, *turb3d*, and *applu*) and integer (*ijpeg* and *compress*) benchmarks. In all cases the input data sets have been reduced from the SPEC distribution to reduce running time. In addition we measure *alvinn*, a floating point program from SPEC92, which we have modified to read its input data from its data segment rather than a file. This reduces time it takes *alvinn* to reach a computational steady state, which is the behavior of interest. *Mandel* is a small program that generates a color bitmap of the Mandelbrot set and writes the result to a file [Gil].

The results can be seen in Figure 2. All benchmarks were compiled with DEC's C or Fortran

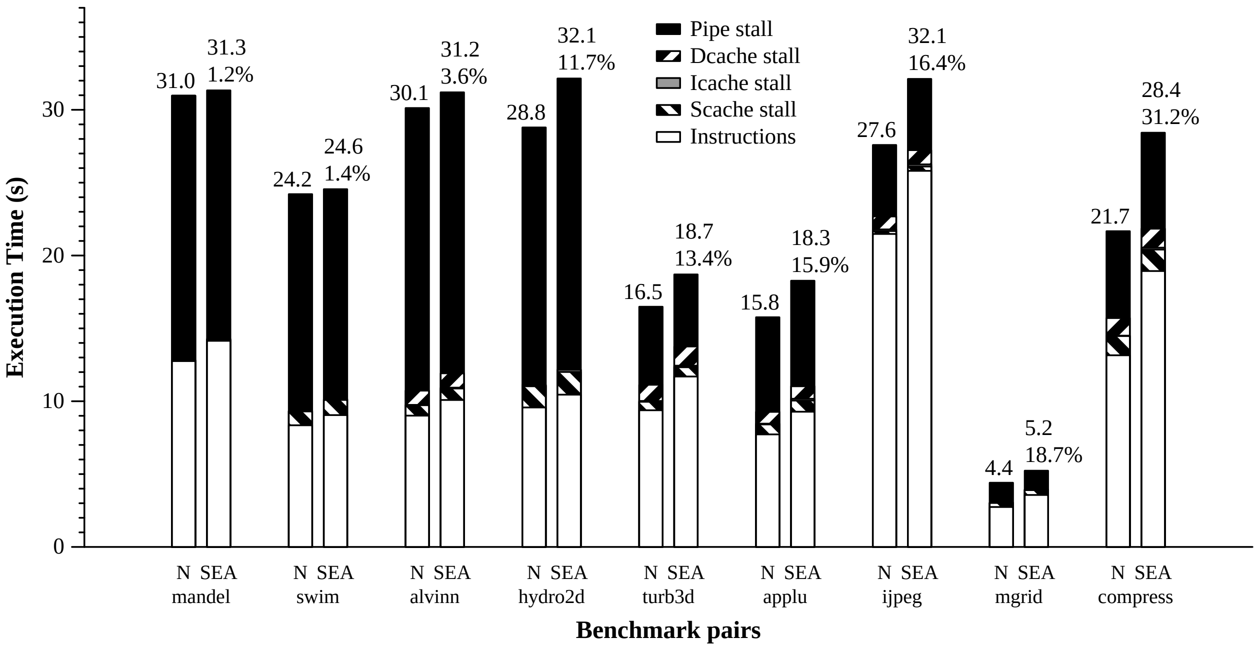


Figure 2: Execution time for several benchmarks (compilied at the highest optimization level), running native (N) and running under SEA (SEA) are presented. Aggregate time for each pair is broken down among stall for pipeline dependencies, primary data (D) cache refill, primary instruction (I) cache refill, and unified on-chip secondary (S) cache refill, and instruction execution. The exact running time of each benchmark pair is given above the bar, and the change from native execution to SEA execution is given with the SEA value.

compiler at the highest level of inter-procedural optimization and chip-specific code scheduling flags. Having the compiler schedule the code for the 21164 makes the overheads of the SEA runtime system more apparent because the compiler's chip-specific scheduling is superior to the chip specific schedule done by the translator. In order to more fully expose the overhead of the SEA runtime system, no feedback information was provided, and conditional branch speculation was turned off. Because SEA can optimize data-dependent branches and loop limits, we expect that its performance using feedback information would be better relative to a compiler that had access to the same information.

Several things are interesting about the data in Figure 2. The first is that the overhead of our SEA implementation is surprisingly low—only 1–3% for three of the nine applications. While our benchmarks to not stress the memory system, our SEA implementation adds little overhead (less than 10%) to memory system costs. SEA also demonstrates little interference with the memory mapping hardware of the machine. SEA's memory and mapping requirements scale (moderately) with application size, not with application data sets, so we conclude that the architectural cost of an SEA is acceptably low.

Overhead due to SEA support functions (e.g., translator and dispatcher) is shown in Table 1. It too is surprisingly low, indicating that the optimizations developed for Shade and Embra translate well to SEAs.

In all cases, the increased running time on the SEA is due to increased instructions. Integer programs like *ijpeg* and *compress* have more procedure returns than the floating point programs, and so suffer from larger instruction overheads. The comparison of IPC for native and SEA execution in Table 1 indicates that the SEA runtime is not finding unused issue slots to hide its overhead instructions. The SEA bookkeeping code actually reduces the IPC of most of the benchmarks. While the SEA runtime increases the IPC of *mandel*, Figure 2 shows a definite increase in the amount of time spend executing instructions. This implies that the SEA bookkeeping code for *mandel* is efficiently scheduled, but the SEA cannot find room in the compiler output to schedule its overhead instructions.

Currently, the SEA runtime handles register indirect jumps (e.g., those present at procedure returns) by using the register that holds the new PC value to index into a hint table. The hint table associates program PC values with the translation cache address for that code's translation, if one exists. Because the table is just a hint, the PC value must be checked against the PC value stored in the table. Along with the table indexing overhead, this amounts to approximately 15 instructions. We are investigating ways, e.g., by using a return stack, to lower this cost.

It is not surprising that instruction overheads dominate in this experiment because the compiler schedules its code well, and since our SEA is not speculating through basic blocks, its scheduling opportunities are limited, and bookkeeping costs are not amortized over large translation units.

4 Software Simultaneous Multithreading

Modern processors feature multiple functional units that work in parallel. Although most of these units are fully pipelined, they often sit idle due to register dependencies, i.e. one instruction produces a register value needed by a downstream instruction. It may be that the functional unit for the downstream instruction is available, but the instruction cannot execute until the upstream instruction completes.

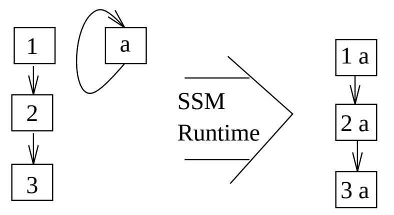
On statically scheduled processors (like the Alpha 21164 and the MIPS R8000), register dependencies cause the processor to stall in the issue stage. The problem of a stalled pipeline is exacerbated on these processors by their multiple issue capability. The Alpha 21164 can issue two integer instructions per cycle, so each stall cycle (possibly) prevents two instructions from executing.

On dynamically scheduled processors (like the MIPS R10000, and the UltraSPARC), register dependencies lengthen the functional unit queue length. On these processors, it might be possible to find a non-dependent instruction further downstream with no register dependencies, and use the functional unit to execute that instruction while the dependent instruction sits on a queue. However, queued instructions use valuable chip resources (like physical registers). Also, the limited lookahead of these processors, coupled with the necessity of branch prediction, makes finding useful, non-dependent instructions difficult.

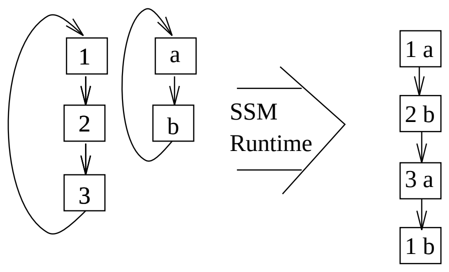
SSMT addresses this problem of pipeline underutilization by dynamically merging instruction streams from different processes. We have implemented SSMT by modifying the dynamic translator of our software-extended Alpha 21164. We discuss the reduced locality of merged programs, the new code scheduling tasks needed for effective program merging, the problems of register allocation and naming, the operating system tasks performed by the SSMT runtime, and finally we discuss the implementation and measure the performance of our prototype.

4.1 Reduced locality of merged programs

In order to merge two processes, the SEA translator needs to keep track of PC pairs instead of a single PC. Code for PC pairs is read, scheduled, emitted and executed as a unit. This change is significant because the locality of PC pairs is lower than that of individual PCs. Reduced locality



(a) Loop smearing



(b) Loop meshing

Figure 3: Merging basic blocks from different processes. One process has blocks 1,2,3, the other has blocks a and b. Loop smear spreads iterations of a loop across multiple translation units, while loop meshing causes the entire cross-product of basic blocks to be merged.

increases the amount of bookkeeping instructions that get executed, and increases the amount of code generated by the translator. Reduced locality has two forms—*loop smearing* and *loop meshing.*

Loop smearing (Fig. 3a) occurs when one process is executing a loop while the other process executes straight line code. Since the merged contents of each loop iteration is different, each loop iteration gets smeared out across the translation cache.

Loop meshing (Fig. 3b) occurs when two processes are each in a loop. When the number of basic blocks in each loop are relatively prime, then if there are enough loop iterations, all combinations of the basic blocks will be emitted. Even if the numbers are not relatively prime, a large fraction of the cross-product may be emitted into the translation cache. The effect is analogous to inter-meshing gears.

A less fundamental, but still annoying, consequence of merging programs is that the overhead of register indirect jumps increases when PC pairs are used to index the hint table. More instructions are needed to compute the hash function and to do the tag check.

4.2 Code scheduling

The benefit of SSMT comes from its ability to schedule two independent instruction streams so pipeline stalls are avoided and latencies overlapped with other latencies or with execution. The prototype SSMT runtime system contains a simple, linear time, greedy code scheduling algorithm. When register dependencies or other pipeline hazards are detected, the instruction scheduler uses instructions from the other thread to pack the pipeline. Pipeline latency values are stored in a table, so adding support for a new Alpha pipeline consists of filling in the table.

For dynamically scheduled processors, it is not only important to schedule for pipeline stalls, but also for the memory system stalls. Computation from one process should be scheduled to overlap with memory access latency from the other process. While dynamically scheduled processors try to hide memory latency within a thread, there is a significant challenge to finding useful instructions. These can be provided to the hardware by the SEA instruction scheduler.

There is also a notion of fairness in scheduling code. Different applications have different sized basic blocks, and benefit from different degrees of branch speculation. The merger can attempt to balance the number of instructions fetched from each thread for each translation unit, so the applications run at roughly the same rate. Using estimates of memory access time would probably result in a fairer schedule.

4.3 Register allocation and naming

There are two important issues involving registers, the first is how the register specifiers in a binary are mapped onto machine registers, and the second is the register requirements of the runtime system itself.

In order to map program registers onto physical registers, the SSMT runtime supports per-processor register maps. If the host architecture has enough registers to support both binaries (e.g., an architecture with 64 integer and 64 floating pointer registers could accommodate 2 Alpha binaries), then the register map is simple, e.g., one process gets registers 0–31, the other 32–63. The SSMT translator then uses the maps to performs register renaming at translation time.

4.4 Operating system tasks

Since the translator merges two programs, the SSMT runtime needs to perform certain operating system tasks, such as address space layout, fault isolation, access to shared state, and system calls.

Address space layout. The SSMT runtime controls the address space layout. Since the goal of SSMT is performance, applications are mapped into disjoint areas of the address space so that (unlike Shade and Embra) runtime address translation is not needed.

Fault isolation. Most operating systems guarantee that independent processes are fault isolated from each other. Programs that load from outside their mapped segments (i.e., text, heap, stack, mmaped regions), are generally sent a signal by the operating system. Running under SSMT, this signal might not be generated. More seriously, a program that stores outside its mapped segments might not be sent a signal, but worse, it may corrupt the process with which it is being merged. While a large class of applications would be willing to trade this level of fault isolation for performance, some would not. More fault isolation could be obtained by architectural support, or by sandboxing [WLAG93] stores so that it would not be possible for a buggy or malicious process to corrupt the data structures of the other process.

Sharing. Having processes share an address space simplifies access to shared state. Processes can request shared memory regions from the SSMT runtime. Access to these regions can be very efficient if fine grained access control is not needed. If such control is needed, sandboxing or the use of virtual memory protection (depending on the sharing granularity) can be used. Special move instructions could also allow applications to share data at the register level.

System calls. The prototype SSMT runtime simply forwards system calls to the underlying operating system. Therefore, both processes block when either does a blocking system call. So long as the operating system exports some mechanism for dealing with blocking system calls—i.e., non-blocking versions of the same calls, or scheduler activations [ABLL91]—the runtime system can execute blocking calls in a non-blocking way. If the SEA is being done by the operating system, then the translator can be informed that one kernel thread is blocked, and the translator will continue running only the other thread.

4.5 Implementation status of SSMT

Most of the discussed features of SSMT have been implemented in the prototype, but there are some exceptions. The current loader is primitive in that it does not relocate dynamically linked binaries—programs are statically linked at disjoint address ranges. Code is currently not scheduled for fairness, i.e. programs can run at very different rates. The prototype does no per-process recompilation—code is not currently rescheduled across basic blocks. Finally, while different processes benefit from different levels of branch speculation, it is currently uniformly performed for both processes.

4.5.1 Merging policies

The policy space for the merger is large and, since there are no run-time systems whose purpose is performance, there is a no literature to directly guide policy decisions. Although the merger has many capabilities, we are still tuning its performance as we come to understand its behavior.

One SSMT policy question is whether to merge all code or to only merge code that is heavily executed. The aggressive approach has an advantage in simplicity, but the opportunistic policy addresses loop smear. In practice, the opportunistic scheme has a small advantage.

The other fundamental SSMT policy question is how aggressive to make the fetch unit of the translator. Currently, the translator aggressively speculates through unconditional branches, and a variable number of conditional branches.

In order to get information for policy decisions, the SSMT runtime uses feedback information. We use ATOM to instrument our executables to measure performance and write information from the execution into a file, which is read by the SSMT runtime system. Currently, this file contains a single branch prediction bit for each program branch, and a single bit for each instruction indicating if it is executed heavily. We currently do not use feedback for data-dependent optimizations.

Our ATOM tool also measures jump behavior. If a certain register indirect jump almost always has the same destination (which is rare because most indirect jumps are procedure returns), its translation can be special cased from 26 instructions to 4 instructions. We are currently looking for ways to integrate this optimization with jumps that are not perfectly behaved, but which transfer control to one site very frequently.

The results shown in this section are for different translator policies. The merger always schedules code for the 21164 pipeline, and always fetches through seven unconditional branches. We vary the number of conditional branches that are fetched through, and are currently looking for a single optimal solution. An opportunistic merging scheme is used unless indicated.

4.6 Experimental evaluation

We believe that SSMT is an interesting application of SEA, and we wanted to investigate the feasibility of the technique. Since SSMT decreases locality and increases working set size in order to reduce register dependencies, it is not likely to be useful for the current generation of architectures. Our goal in building the prototype was to see if any performance wins for current architectures were possible, and to measure the architectural effects of SSMT to determine if it might be a good idea for future architectures.

The limited issue width of the 21164, two integer slots, one general purpose floating point slot, and one floating point multiply slot, represents a challenging architecture for SSMT. There are limited issue opportunities, and significant issue restrictions on some instruction types and instruction pairs.

The static scheduling of the 21164 shows off the code scheduling algorithms of the SSMT runtime, but we believe that the SSMT runtime would also be beneficial (perhaps more so) for dynamically scheduled processors. Studies [Fis93] have shown that software techniques which increase the mix of non-dependent instructions in the hardware's limited instruction window improve processor utilization.

Benchmark	Time		
	Sec	16 int reg	SEA
mandel	38.6	0.008%	-3.6%
compress	24.9	2.7%	16.1%
swim	46.6	6.0%	2.8%
alvinn	25.8	9.9%	-6.7%
ijpeg	31.1	30.0%	7.5%

Table 2: Increase in execution time due to limiting the integer register sets of some benchmarks compiled with gcc. The “SEA” column reports the additional overhead of running the reduced integer register binary on an SEA. The negative values indicate a performance improvement

4.6.1 Register set size

Program merging from Alpha binaries to the Alpha 21164 has a fundamental problem because the binaries are compiled to use all 64 architecturally visible registers (32 integer and 32 floating point). This creates heavy contention for registers. In order to get performance wins on the 21164, this contention must be addressed.

Since the machine registers are over committed, register specifiers for different processes are mapped to the same physical register, e.g., process 0 might keep its value of a0 in a0, while process 1 might keep its value of t3 in a0. Only one process can keep its register value in the register, while the other process must load its register value from memory.

While there are optimizations to reduce the number of register saves and restores, a significant number remain. The issue width of the 21164 is not sufficient to hide these instructions. In order to experimentally investigate the potential of SSMT, we reduced register contention by limiting the number of integer registers each program was compiled with from 32 to 16. Our hypothesis is that running two 16 integer register binaries on a 32 integer register machine is similar to running two 32 integer register binaries on a 64 register machine. This setup also stresses our register renaming logic as both binaries are compiled to use the same 16 registers. We allow each program to use all 32 floating point registers because contention for the floating point register file is smaller, and we wanted to interfere with the compiler as little as possible. Finally, we note that since we do not have source code for certain important system libraries (like most of libc), our register partitioning is approximate.

The register maps reflect the register partitioning of the integer register file. Both binaries make small use of a certain range of temporary registers. One binary uses the registers specified in its program, the other binary has its register specifiers translated to the set of infrequently used temporary registers.

There is still some contention for integer registers, and potentially considerable contention for floating point registers. When there is a register conflict (e.g., both programs use floating point register 3), the translator chooses a different register for one of the programs (e.g., floating point register 5). While the translator looks for dead registers (see Section 4.3), it will spill and restore a live register if it must. If a register is used read-only, it will not be written at the end of the translation, and if a register's first appearance is as a destination, its value will not first be loaded.

4.6.2 The measured impact of limited integer registers

Before evaluating the performance of SSMT on integer register limited binaries, we want to determine how reducing the available integer registers affected the programs we studied.

The only compiler we had access to which allows fine grained control over register use is gcc. This is unfortunate since gcc does no chip specific scheduling for the DEC 21164, so the rescheduling done by the SSMT runtime is more efficient than the original schedule, plus the benefit from overlapping latencies.

Since the Gnu fortran compiler is not yet supported on the Alphas, we use SUIF [HAA$^+$96] to translate the fortran benchmarks into C, which we compiled with gcc. The resulting binaries are less efficient than versions compiled with a Fortran compiler, but we believe that they are inefficient in a way that penalizes the SSMT runtime. For example, when *swim* is compiled with the Fortran compiler, it spends 60.6% of its execution time stalled on register dependencies. When it is compiled by gcc with Fortran support libraries, it only spends 48.0% of its execution time stalled on dependencies. This makes sense because better compiler technology can not remove basic register dependencies, it can only eliminate instructions which mask those latencies.

Table 2 shows the performance impact of moving from 32 integer registers to 16 integer registers (using gcc at the highest level of optimization) for the benchmarks we studied in depth.

The impact of limiting the number of available registers differs widely for different benchmarks. *Mandel* and *compress* do not make heavy use of integer registers, while *alvinn*, and especially *ijpeg* benefit from larger integer register sets.

While the performance differences are not always large, 16 integer register binaries often have a higher percentage of loads and stores. While this spill code usually has good cache locality, it does add to data cache pressure. Additionally, stores can not dual issue on the 21164, so they are particularly poor instructions for the translator. As partial compensation, load instructions have a delay slot which the translator can use, but there are odd issue restrictions (arising from a structural hazard for the data cache ports) regarding loads and stores that occur close together in time. We believe that on balance, 16 integer register binaries represent a pessimistic case for SSMT.

Finally, the SEA "overheads" reported in Table 2 are actually significant performance gains for two applications. Our evaluation of base SEA (Figure 2), did not demonstrate these gains because the base SEA results were generated without using feedback information for branch prediction. Additionally, we are showing performance wins for applications compiled with gcc which does not do chip-specific instruction scheduling.

It is important to note that the gains shown for SSMT (in Section 4.7), are not solely due to fixing the code scheduling of gcc. First, the gains reported here are smaller than the gains for SSMT, and second our measurements of SSMT show that instruction scheduling is only part of the win.

4.7 The measured performance of SSMT

SSMT on the 21164 slowed down almost all application pairs we investigated. However, performance was within a factor of 2.5 of the native execution time for almost all pairs indicating that the average case was not pathologically bad. This agrees with our intuitions that the 21164 architecture is not a good candidate for SSMT.

However, certain aggressive merging policies were able to show performance gains for certain application pairs. By studying some of these cases, and some cases where performance was reduced,

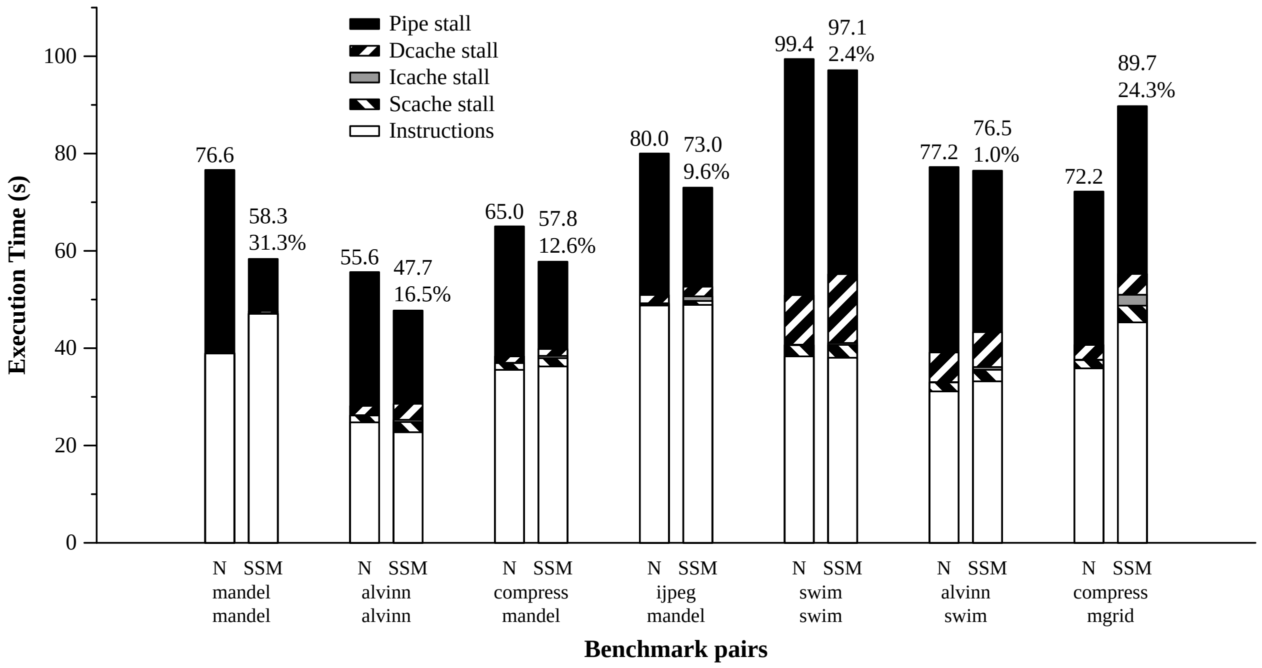


Figure 4: Execution time for several 16 integer register benchmark pairs running native (N) and running under SSMT (SSM) are presented. Aggregate time for each pair is broken down among stall for pipeline dependencies, primary data (D) cache refill, primary instruction (I) cache refill, and unified on-chip secondary (S) cache refill, and instruction execution. The exact running time of each benchmark pair is given above the bar, and the change from native execution to SSMT execution is given with the SSMT value.

we can understand the properties of SSMT, and have a persuasive basis for arguing that it will be useful on more aggressive architectures.

A range of application performance is presented in Figure 4. SSMT increases cache pressure more than base SEA did, which is no surprise given that the machine must now support the sum of the application working sets. The overhead due to SSMT data structures is small.

SSMT greatly increases the instruction cache stall of the measured programs, though the magnitude of that stall remains small. This indicates that our translation policies work well for programs with small instruction footprints, and it indicates that SSMT might be able to reduce instruction cache misses on large binaries, so long as the working set of the program or program pair can fit in the translation cache.

The overhead due to SSMT support procedures (presented in Table 3), is also slightly larger than the base SEA case, but it is 3% or less for most of the benchmarks we studied.

In general, SSMT was able to obtain performance wins by reducing the amount of pipeline stall. In almost every case, it increased the amount of time spent executing instructions. A notable exception is the *alvinn* pair. The reduced amount of instruction runtime for this pair, coupled with the data in Table 2 indicates that the SSMT runtime is not only efficiently hiding its own overhead, it is effectively rescheduling *alvinn*'s code. This can be seen from the increased IPC as reported in Table 3.

SSMT's biggest performance win (31% throughput increase) was for the *mandel* application pair. The SSMT translator is effectively scheduling the merged code to reduce pipeline stalls, but

Benchmark pair		IPC		SSMT Runtime
		Native	SSMT	time
mandel	mandel	0.57	1.21	0.39%
alvinn	alvinn	0.65	0.81	3.12%
compress	mandel	0.63	0.83	0.82%
ijpeg	mandel	0.73	0.88	2.93%
swim	swim	0.62	0.74	0.48%
alvinn	swim	0.63	0.70	3.38%
compress	mgrid	0.66	0.64	6.77%

Table 3: IPC and SSMT runtime system overhead for several 16 integer register benchmark pairs. Overhead due to SSMT support routines is low, but SSMT adds extra instructions to the workload. How these instructions are scheduled influences the IPC.

unlike the *alvinn* pair, it is still adding a significant number of overhead instructions. But as we saw in Section 3.4, the overhead instructions for *mandel* can be efficiently scheduled. The combination of these effects yields an impressive (greater than 2x) increase in IPC as reported in Table 3.

While SSMT shows benefit from chip-specific scheduling, the wins reported in Table 2 are much smaller than those reported in Figure 4. The main cause of the performance win in every case is reduced pipeline stall. Even if the original binaries were compiled with chip-specific scheduling all pairs that show a performance win, except the pair *alvinn* and *swim*, would continue to show performances win.

Given the data in Table 2, the performance gains for the *mandel* pair and the *alvinn* pair is still a gain when measured against a binary (compiled with gcc) that can use all 32 integer registers. It is encouraging that for the 21164 there are some cases for which SSMT provides the most efficient execution model.

SSMT relies on the availability of pipeline stall to gain its wins. As seen in Figure 2 pipeline stall seems plentiful on the 21164, even using advanced fortran compilers, and wider issue machines will increase this availability [KOHW$^+$96]. However, floating point intensive applications generally spend more time waiting for register interlocks because the latency of floating point operations is longer (4 cycles latency for a floating point add on the 21164 as opposed to 1 cycle for most integer instructions) making them more suited for SSMT.

We present the *compress* and *mgrid* pair as representative of an application pair that does not benefit from SSMT. This pair does not see a reduction in pipe frozen time, indicating a failure of our code scheduling algorithm. Additionally, the increased time spent in instruction execution indicates that the SSMT is adding too many instructions, and the decreased issue rate indicates that it is not scheduling these overhead instructions efficiently.

It is not a coincidence that three of the six performance wins presented are applications merged with another copy of themselves. Because our prototype currently does not schedule code for fairness, because the benchmarks do not run for exactly the same amount of time, and because the benchmarks run for a relatively short period of time, there is often significant load imbalance when running under SSMT. One process finishes, and up to 40% of the combined running time is consumed by the slower process running to completion by itself. When a process is merged with another instance of itself, both processes run together. Running the applications on their full data sets would help alleviate this problem, and we believe that future work in our code scheduling algorithms will make this less of an issue.

While SSMT is not appropriate for todays architectures, our measurements indicate that it is effective at reducing pipeline stall, and that it can be effective at chip-specific scheduling. We believe that the importance of these benefits will increase as processor technology progresses.

5 Discussion

In this section, we discuss some architectural features that would benefit SSM, and we discuss the architectural vision enabled by SSM. We have already mentioned how SSM would benefit from a wider issue machine, because it could schedule its bookkeeping instructions in the unused slots.

Register set size For SSM to be profitable, there must either be a larger number of physical registers than are used by most binaries, or instruction issue slots must be plentiful. There are issues in making a larger register set architecturally visible. The first is binary compatibility and the second is instruction encoding. SEA directly addresses binary compatibility as binaries are not directly executed on the processor. However, if an application is particularly poorly suited for SEA, we believe that binary translation technology [SCK$^+$93] has progressed to the point that translating a RISC instruction set between encodings can be simply done, and the results are efficient.

The issues surrounding instruction encoding are more substantial, but are beyond the scope of this paper. Alternatives include alternate encodings if the number of physical registers is 64, or 64 bit instructions for larger register files.

If there are more hardware registers available than are required to run a given application (or application pair), the translator can use simple heuristics to eliminate memory accesses.

Branches Our measurements for the change in branch misprediction rates for programs run both under SEA and SSM vary widely, from small improvements, to a factor of 2 degradation. However, if the base architecture had branch prediction bits in the instruction encoding, the translator could pass its prediction information down to the hardware. We are also investigating ways of collecting branch direction information dynamically, so the translator could significantly extend the hardware's prediction capabilities.

In the common case, SSM translations require the resolution of multiple branch conditions, which is potentially expensive to the pipeline. Currently, other performance issues are more important, and with aggressive branch speculation, the multiple resolution overhead might be a small part of the execution time of a translation unit. Still, architectural support in the form of a jump table instruction which evaluates multiple conditions might be useful.

Floating point latencies SSM exploits the multi-cycle latency of floating point and some integer instructions. While these latencies have been decreasing, this has been offset by wider issue widths (e.g., on the Alpha 20164 fp latency is 6 cycles and 6 missed integer instruction opportunities. On the Alpha 21164, fp latency is 4 cycles, and 8 missed integer instruction opportunities). While the availability of transistors might eliminate this latency on future processors altogether, memory reference latency is much harder to eliminate. We believe that our SSM algorithms can be tailored to increase the throughput of a dynamically scheduled superscalar processor by overlapping memory access latency with computation.

User interface While the work presented in this paper consists of executables that are specially prepared, launched in tandem and run to completion, more flexible user level interfaces are possible. These include augmenting the operating system command shell (such as the UNIX *sh* shell) with a special pipe character; modifying the scheduler of a user level threads package; and implementing a set of shared memory macros (e.g. ANL). As mentioned earlier, it is also possible for the operating system scheduler to provide the merging service.

Impact on processor design In designing ILP processors, architects need to balance hardware complexity with reliance on sophisticated software (e.g., advanced compilers). The current generation of dynamically-scheduled processors represents a hardware-intensive solution, while VLIW processors represent a software-intensive solution. Software-extended architectures might allow architects to revisit this fundamental tradeoff. An SEA could ensure binary compatibility for user programs, and can greatly simplify processor design by performing register renaming, dependency checking, and code scheduling. This could allow processor designers to simplify their design and make more radical changes in between chip version.

We are not sure what the future, or ultimate utility of software-extended architectures will be, but it seems like interposing a layer of software at the architecture level opens up new possibilities. In this paper we have discussed how an SEA interacts with the processor architecture, the compiler, and the operating system. In almost every case, SEA either provides new functionality, or it implements some functionality at a more natural level of abstraction than it was originally provided.

6 Conclusions

In this paper we have demonstrated that a software-extended architecture can be implemented with very little performance overhead. Our implementation of a software extended 266 Mhz Alpha 21164 adds only 1%—30% to the running time of a program. We have also demonstrated that a software-extended Alpha 21164 can be used to improve system throughput by supporting software simultaneous multithreading. SSMT trades decreased locality and increased working set size for increased pipeline utilization. While this is of limited value for applications on the Alpha 21164, our data indicates that this tradeoff is appropriate for next generation machines.

This paper establishes the viability of software-extended architectures, but it also leaves a number of open research questions. The key ones are (1) what are good policies for scheduling merged programs, (2) can software-extended architectures enable wider acceptance of VLIW, or allow superscalar architects to simplify their design and concentrate on increasing clock rate and adding more functional units, and (3) what other performance benefits and new capabilities can be obtained using software-extended architectures. We expect to investigate these issues in the near future.

7 Acknowledgments

We would like to thank Eddie Kohler for doing dead register analysis, and Bert Halstead and DEC's Cambridge Research Lab for letting us use their machines.

References

[ABLL91] T. E. Anderson, B. N. Bershad, E. Lazowska, and H. M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. In *Proc. Thirteenth ACM Symp. on Operating System Principles*, page 95, Pacific Grove, CA, Oct 1991.

[ATLLW96] Ali-Reza Adl-Tabatabai, Geoff Langdate, Steven Lucco, and Robert Wahbe. Efficient and language-independent mobile programs. In *Proc. of PLDI*, may 1996.

[CK94] Robert F. Cmelik and David Keppel. Shade: A fast instruction set simulator for execution profiling. In *SIGMETRICS*, 1994.

[CSB96] Bradley Chen, Michael D. Smith, and Brian N. Bershad. Morph: a framework for platform-specific optimizations. Technical Report White paper, Harvard University, March 1996.

[Dig] Digital. Fx!32: x86 win32 compatibility on aplha. Technical Report http://www.service.digital.com/fx32/.

[Eng96] Dawson R. Engler. vcode: a retargetable, extensible, very fast dynamic code generation system. In *Proc. Programming Language Design and Implementation*, 1996.

[ER] John Edmondson and Paul Rubinfeld. An overview of the alpha axp(tm) microarchitecture. In *slides*.

[Fis81] J. A. Fisher. Trace scheduling: a technique for global microcode compaction. *IEEE Transactions on Computers*, 30:478–490, July 1981.

[Fis93] Joseph A. Fisher. Global code generation for instruction-level parallelism:trace scheduling-2. In *HP Laboratories Technical Report HRL-93-43*, June 1993.

[Gil] Frode Gill. http://www.krs.hia.no/ fgill/mandel.{html,c}.

[Gos95] James Gosling. Java intermediate bytecodes. In *Proceedings of ACM SIGPLAN Workshop on Intermediate Representations*, March 1995.

[GYCS96] Nicolas Gloy, Cliff Young, J. Bradley Chen, and Michael D. Smith. An analysis of dynamic branch prediction schemes on system workloads. In *Proc. 23th Annual Symposium on Computer Architecture*, pages 12–22, May 1996.

[HAA$^+$96] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.W. Liao, E. Bugnion, and M. S. Lam. Maximizing multiprocessor performance with the suif compiler. In *IEEE Computer*, December 1996.

[Jon93] Michael B. Jones. Interposing agents: Transparently interposing user code at the system interface. volume 27, pages 80–93, Dec 1993. 14th ACM Symposium on Operating Principles.

[KOHW$^+$96] Basem A Nayfeh Kunle Olukotun, Lance Hammond, Ken Wilson, , and Kunyung Chang. The case for a single-chip multiprocessor. In *Proceedings of ASPLOS-VII*, Oct 1996.

[LS95] James Larus and Eric Schnarr. Eel: Machine independent executable editing. In *Proceedings of PLDI*, June 1995.

[MEP96] Po-Yung Chang Marius Evers and Yale N. Patt. Using hybrid branch predictors to improve branch prediction accuracy in the presence of context switches. In *Proc. 23th Annual Symposium on Computer Architecture*, pages 3–12, May 1996.

[MS96] Larry McVoy and Carl Staelin. lmbench: Portable tools for performance analysis. In *USNIX*, Jan 1996.

[Pat85] D. A. Patterson. Reduced instruction set computers. *Communications of the ACM*, 28(1):8–21, January 1985.

[RBH$^+$95] Mendel Rosenblum, Edouard Bugnion, Stephen A. Herrod, Emmett Witchel, and Anoop Gupta. The impact of architectural trends on operating system performance. In *SOSP*, 1995.

[RS94] Rahul Razdan and Michael D. Smith. High-performance microarchitectures with hardware-programmable functional units. In *Proceedings of the 27th Annual IEEE/ACM Intl. Symp. on Microarchitecture*, pages 172–180, November 1994.

[SCK$^+$93] R. L. Sites, A. Chernoff, M. B. Kirk, M. P. Marks, and S. G. Robinson. Binary translation. *Communications of the ACM*, 36(2):69–81, February 1993.

[SE94] Amitabh Srivastava and Alan Eustace. Atom: a system for building customized program analysis tools. *SIGPLAN Notices*, 29(6):196–205, June 1994.

[Smi91] Michael D. Smith. Tracing with pixie. Technical Report Memo from Center for Integrated Systems, Stanford University, April 1991.

[SPE95] SPEC. Spec cpu 95 benchmark suite. In *System Performance Evaluation Cooperative*, 1995.

[TEE$^+$96] Dean M. Tullsen, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, and Rebecca Stamm. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 23nd International Symposium on Computer Architecture*, May 1996.

[TEL95] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd International Symposium on Computer Architecture*, June 1995.

[Wal86] David W. Wall. Global register allocation at link time. In *Digital research report 86.3*, Oct 1986.

[Wal91] David W. Wall. Limits of instruction-level parallelism. In *ASPLOS-IV*, pages 176–189, Santa Clara, California, 1991.

[WLAG93] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient software-based fault isolation. In *SOSP14*, pages 203–216, December 1993.

[WR96] Emmett Witchel and Mendel Rosenblum. Embra: Fast and flexible machine simulation. In *The proceedings of ACM SIGMETRICS '96: Conference on Measurement and Modeling of Computer Systems*, 1996.

[YP92] Tse-Yu Yeh and Yale N. Patt. Alternative implementations of two-level adaptive branch prediction. In *Proc. 19th Annual Symposium on Computer Architecture*, pages 124–134, may 1992.