Automatic Synthesis of High-Speed Processor Simulators

Martin Burtscher and Ilya Ganusov
Computer Systems Laboratory, Cornell University
{burtscher, ilya}@csl.cornell.edu

Abstract

Microprocessor simulators are very popular in research and teaching environments. For example, functional simulators are often used to perform architectural studies, to fast-forward over uninteresting code, to generate program traces, and to warm up tables before switching to a more detailed but slower simulator. Unfortunately, most portable functional simulators are on the order of 100 times slower than native execution. This paper describes a set of novel techniques and optimizations to synthesize portable functional simulators that are only 6.6 times slower on average (16 times in the worst case) than native execution and 19 times faster than SimpleScalar's sim-fast on the SPECcpu2000 programs. When simulating a memory hierarchy, the synthesized code is 2.6 times faster than the equivalent ATOM code. Our fully automated synthesis approach works without access to source/assembly code or debug information. It generates C code, integrates optional user-provided code, performs unwanted-code removal, preserves basic blocks, generates low-overhead profiles, employs a simple heuristic to determine potential jump targets, only compiles important instructions, and utilizes mixed-mode execution, i.e., it interleaves compiled and interpreted simulation to maximize performance.

1. Introduction and Motivation

Neither industry nor academia can afford the time and money to fabricate a new processor every time they want to test a new idea. Instead, they resort to simulators, which can relatively easily be modified to model novel architectural features. For example, functional simulators are frequently used to generate traces, to prototype and test ideas, and to fast-forward over uninteresting code while optionally warming up various structures before switching to a cycle-accurate simulation mode. Interpretation-based functional simulators are relatively simple to port but can be slow. The faster translation-based functional simulators either work only on machines of a specific instruction set architecture (ISA), require access to source or assembly code, or perturb the original binary, which can change memory addresses and pointer values.

SyntSim, the functional-simulator synthesizer presented in this paper, combines the portability and non-perturbation of true interpreting simulators with the speed of translators while only needing access to the executable. In fact, it can run executables devoid of any symbol or debug information, which is not even possible with binary-translation tools like ATOM [11, 38] and Spike [18]. Because SyntSim is written in C and generates C code, it can run programs on most target architectures for which a C compiler exists. Moreover, it is straightforward to incorporate user-provided code into SyntSim, for example, to simulate a cache or a branch predictor.

SyntSim simulators execute the SPECcpu2000 programs with the reference inputs only 6.6 times (geometric mean) slower than native execution, even when accounting for the synthesis and compilation time. The slowest program (gcc) experiences a slowdown of 16.0 while the fastest program (mcf) is simulated at half its native speed. These high speeds are the result of a set of optimizations that are the topic of this paper.

Due to its speed, SyntSim is a great choice wherever functional simulators are needed. For example, many researchers use statistical [9, 15, 26, 29, 41] or profile-driven [25, 35, 37] sampling whenever the entire program execution cannot be simulated, both of which require fast-forwarding. SyntSim is ideal for this purpose. Even with early SimPoints [35], it takes SimpleScalar's *sim-fast* [6] an average of 1.9 hours to fast-forward and then another 75 minutes for the detailed cycle-accurate simulation per SPECcpu2000 program on our system (Section 4.1). With SyntSim, the fast-forwarding time is reduced to 16 minutes, doubling the overall simulation throughput. Note that SyntSim can warm up caches and other CPU components while fast-forwarding. In fact, it can warm up a memory hierarchy 2.6 times faster than ATOM. Of course, SyntSim can also be used for all other purposes for which functional simulators are utilized, for example, to rapidly generate the execution traces needed to drive certain cycle-accurate simulators [5, 24].

SyntSim's fast execution is the result of a judicious combination of compiled and interpreted simulation modes as well as several optimizations, including unwanted-code elimination, NOP removal, label minimization (preservation of basic blocks), inlining, PC update minimization, and the hardcoding of constants, register

Proceedings of the 37th International Symposium on Microarchitecture (MICRO-37 2004)
1072-4451/04 $20.00 © 2004 IEEE

specifiers, and displacement/immediate values. Some of these optimizations directly speed up the simulator. Others do so indirectly by enabling the C compiler to perform aggressive optimizations. Before each simulation, SyntSim automatically synthesizes a customized simulator. The generated C code is then compiled and thus can run all or part of the simulation in compiled mode rather than in the slower interpreted mode. The time required to generate and compile the simulator is usually amortized within minutes of running the simulation.

Since SyntSim needs to amortize the synthesis and compilation time, its overall performance depends not only on the length of the simulation but also on the amount of code that needs to be synthesized and compiled. Our results show that it is typically best to aim for running about 99.9% of the dynamic instructions in compiled mode and the remaining 0.1% in interpreted mode. This means that only about 15% to 37% of the static instructions need to be compiled. Running more instructions in compiled mode worsen the overall turnaround time because they essentially do not make the simulation faster but do increase the compilation time significantly. Short simulations prefer lower ratios of compiled to interpreted execution because the resulting shorter compilation times can be amortized more quickly. Note that the decision about which instructions to run in which mode are made statically at synthesis time. SyntSim is not a JIT compiler and does not need dynamic compilation support.

SyntSim allows the user to specify the approximate fraction of the instructions to be simulated in compiled mode. This makes it possible to trade off compilation time for simulation time and to adapt to the capabilities of the compiler and the expected length of the simulation run, all of which we found to impact performance.

Also, we found that a "-O1" compiler optimization level provides the best trade-off between compilation time and simulation speed and that unwanted-code elimination and NOP removal can be performed much more quickly by the synthesizer than by the compiler.

Compiled mode is much faster than interpreted mode because the overhead of decoding instructions (typically multiple nested switch statements in interpreters) and extracting fields from the instruction words is removed. No code is generated for NOPs and unreachable instructions. NOPs can safely be skipped in functional simulations. Since SyntSim only emits labels in the generated C code for actual branch and jump targets, the original basic blocks are preserved, allowing the compiler to optimize and schedule the code more effectively than would be possible if every instruction were a potential branch/jump target. We found that minimizing the emitted labels results in a 43% speedup. In addition, all register specifiers, immediate values, and zero constants for the zero registers are hardcoded to simplify the task of the compiler. In fact, because of the hardcoded register specifiers, the compiler is able to perform constant and copy propagation as well as scalar replacement of each (simulated) register even though the register file is specified as an array. Inlining ensures that there is no call overhead and exposes more potential for compiler optimizations. Since most instructions do not need to know the value of the PC explicitly, SyntSim only updates the PC register where necessary. Furthermore, it exposes user-supplied simulation code to the same kind of optimizations and inlines it, which tools like ATOM cannot do.

The remainder of this paper is organized as follows. Section 2 summarizes related work. Section 3 explains the operation of SyntSim in detail. Section 4 describes the evaluation methods. Section 5 presents performance results and comparisons with other simulators and tools. Section 6 concludes the paper.

2. Related Work

Quite a number of CPU simulators are currently in use. Some are *functional simulators*, that is, simulators that execute programs correctly but do not model any timing behavior or speculation. Simulators belonging to this class include SimpleScalar's *sim-safe* and *sim-fast* [6], AINT [34], as well as the simulators generated by SyntSim. The former are representatives of interpretation-based simulators, which are typically quite portable. In such simulators, every instruction is decoded using a switch statement, each case of which contains code to interpret a particular instruction.

Cycle-accurate simulators, on the other hand, are much slower but accurately model the CPU's internals. This class of simulators includes Rsim [22], Asim [10], and SimpleScalar's *sim-outorder*, *sim-alpha*, and *sim-mase* [3, 27]. There are also *full-system simulators*, which are not restricted to running one program at a time but rather emulate an interactive multi-tasking operating system, including device drivers, interrupt handlers, and privileged instructions. Nevertheless, full-system simulators, including SimICS [31] and SimOS [36], are usually based on an underlying cycle-accurate or functional simulator (or both). Cycle-accurate as well as full-system simulators utilize functional simulators for fast-forwarding purposes, i.e., to quickly execute uninteresting code.

Finally, *binary translation* has been studied extensively in the context of just-in-time compilation [1, 42], FX!32 [17], HP's Aries translator [43], and the BOA architecture [14]. FX!32 [17] is a pure binary translator that converts programs from one ISA to a different ISA while Shade [8] and Embra (used in SimOS) [40] represent simulators that are based on dynamic binary translation. Shade cross-compiles portions of source executables and runs them directly on the host machine, saving translations for reuse to amortize the cost of the compilation. Embra extends Shade's approach to support full OS simu-

lation. It does so by inserting special, highly optimized code sequences into the translated code. Since such simulators are restricted to run on a machine of a predetermined ISA, binary translation-based simulators generally lack the portability of interpretation-based simulators. On the upside, translated code often runs within an order of magnitude in speed relative to native execution and is thus much faster than interpreted simulation. SyntSim provides comparable simulation speeds without restricting the simulation platform to a particular ISA.

Functional simulators that are based on binary translation/instrumentation include ATOM [11, 38], Dyninst [16] and the Augmint multiprocessor simulation toolkit [33]. These simulators insert special code into the original binary to track events of interest (e.g., memory references). The instrumented binary is then executed natively, which provides fast simulation speeds but limits the use of such tools to a specific ISA. Another potential downside of (binary) instrumentation is the perturbation of the original program, in particular the moving of instructions and data to different memory locations, which can change pointer values. Tango [13] is similar to ATOM and Augmint except it instruments code at the source instead of the binary level. SyntSim combines the non-perturbation of interpreting simulators with the speed of binary translation-based simulators.

The MINT simulator [39] was developed as a front-end interpreter for a multiprocessor simulator. It uses a hybrid approach of software interpretation and native execution. Much like interpreters, it operates on a copy of the simulated processor's state in memory. However, it employs a form of code synthesis [32] to speed up instruction execution. When the executable is loaded into the simulator, MINT creates a function for each sequence of code that generates no architectural event of interest. This function is called and natively executed whenever the original sequence of code needs to be executed. Even creating one such function for every individual instruction was found to result in better performance than using switch statements to decode and interpret programs. While MINT avoids the interpretation costs of the instructions captured in the functions, the calling overhead can still be significant. SyntSim does not suffer from this overhead as it inlines all code, which also allows more effective optimization of the code. Moreover, SyntSim's granularity of compilation is not restricted by architectural events.

Krishnan et al. [24] extend the direct execution approach used in MINT to cycle-accurate simulation of superscalar microprocessors. They propose to map the original binary into a simulation ISA that retains the timing information but is lossy in other aspects. They then run an instrumented version of the original binary through MINT to generate a trace of events, which is subsequently passed on to the processor simulator. The simulator needs the event trace to fill in the missing information when executing the simplified ISA instructions. This considerably speeds up cycle-accurate simulations.

Fujimoto and Campbell proposed to use purely static translation for modeling microprocessor behavior [12]. They decompile the original assembly program into a high-level language, instrument it, recompile it, and run it on the host computer. While both Fujimoto's simulator and SyntSim use a decompilation process, SyntSim is different in two key aspects. First, it directly operates on executables while Fujimoto's simulator requires assembly code and cannot deal with executables directly. Second, SyntSim utilizes a combination of static translation and interpretation instead of only static translation.

Other static translation tools that decompile assembly programs into a high-level language include AsToC [19] and XTRAN [20]. However, these two toolsets are primarily designed for software migration and maintenance and, unlike SyntSim, do not support user-provided code.

The University of Queensland Binary Translator (UQBT) [7] uses a similar approach as Fujimoto's simulator. UQBT exploits static translation to inexpensively migrate existing software from one processor to another. UBQT translates the original binary into a special high-level register-transfer language, which is then recompiled into a binary for the target ISA. UBQT inserts special static hooks into the translated binary to interpret untranslated code discovered at runtime. Hence, UQBT shares the idea of mixed-mode simulation with SyntSim. However, it is tailored towards software migration and is not suitable for architectural simulation. Moreover, SyntSim does not require special hooks and allows the user to choose and thus optimize the ratio of compiled versus interpreted simulation. Finally, SyntSim uses C as its intermediate representation, making it more portable.

Similar to SyntSim, Zivonjovic and Meyr's SuperSim [44] uses static decompilation into high-level C code and subsequent recompilation to produce a simulator. However, their approach assumes that every instruction can be a branch target, i.e., is preceded by a label, which we found to substantially decrease the simulation speed and increase the compilation overhead. SyntSim only emits labels for true branch and jump targets.

SimICS is a fast system-level instruction-set simulator [30, 31]. It achieves its high performance by translating code into an intermediate representation to speed up the interpretation. SyntSim eliminates most of the need for interpretation by emitting simulators in a high-level language that are compiled and natively executed. SimICS does not use compiled-mode simulation. Also, SimICS's interpretation routines incur a calling overhead. SyntSim inlines all code to avoid this overhead and to expose the code to more compiler optimizations.

Larus' QPT [4] and EEL [28] tools allow the creation of efficient simulators by rewriting executables and in-

serting instrumentation code. SyntSim differs in that it combines interpreted and compiled mode while QPT/EEL only use compiled mode. Furthermore, in QPT/EEL the simulation and the simulated machine's ISAs have to be the same. While QPT/EEL's algorithm to handle indirect branches is more precise, it sometimes requires dynamic code translation, which makes the system more complex. SyntSim is much simpler and requires no dynamic translation support. EEL can only encapsulate additional code that has been compiled into assembly. SyntSim directly injects the code at the source level, thus exposing it to more compiler optimizations.

3. Synthesis Procedure

This section describes how SyntSim generates a high-speed functional simulator for a given program. It starts out by parsing the executable and computing a checksum. Depending on the command-line parameters, SyntSim reads in optional profile information. Then it selects code for compiled mode, loads the translation table containing the C definitions of each instruction, synthesizes, compiles and runs the customized simulator. The following subsections explain each step in detail. Figure 1 gives an overview over SyntSim's operation.

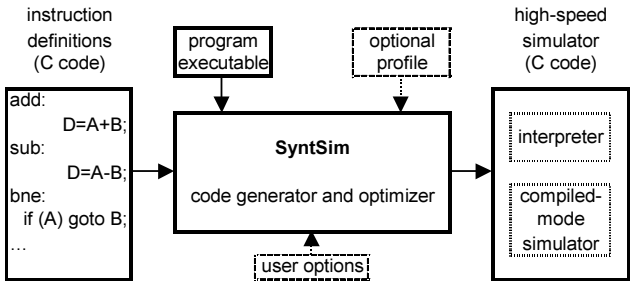

Figure 1: SyntSim's operation.

3.1 Selecting code sequences for compiled mode

Throughout this paper, we adhere to the Alpha convention of calling indirect control-transferring instructions *jumps* and direct control-transferring instructions *branches*. In other words, jump targets are dynamically computed while branch targets are hardcoded in the instructions and are statically known.

Our benchmark programs contain three types of jump instructions, namely jump (JMP), jump subroutine (JSR), and return (RET). All three instructions perform identical operations, that is, they load the PC from a general-purpose register and store the old PC into a general-purpose register (the link register). The only difference between the three instructions is the action performed by the return address stack [2].

In SyntSim, the instructions that can be jump targets represent the entry points for compiled mode. Hence, identifying these instructions plays an important role. SyntSim supports three alternatives: (1) to use no compiled mode, (2) to use built-in heuristics to guess which instructions to include in compiled mode, and (3) to use profile information to determine which instructions to execute in compiled mode.

The first choice, i.e., using no compiled mode, is always available and basically means the built-in interpreter will be used exclusively to simulate a program's execution. This is the slowest mode.

The choice of guessing which instructions to execute in compiled mode is also always available. In this case, SyntSim utilizes the following approach to determine the jump targets and thus the starting points for compiled-mode code sequences. It scans all instructions in the text segment (ignoring other content such as constants) and applies the following heuristics. For each branch-subroutine instruction (BSR) whose target is a valid instruction, it marks the sequentially next instruction as a jump target, but only if the specified link register is not the hardwired zero register. The same heuristic is applied to jump-subroutine instructions (JSR), i.e., the sequentially next instruction is marked as a jump target if the link register is not the zero register. Moreover, for jump (JMP) and jump-subroutine (JSR) instructions, the target is computed based on the hint field in those instructions [2]. If the computed target is a valid instruction, that instruction is marked as a jump target. Together, these heuristics prove quite effective and perform only 32% worse than perfect profile information (see Section 5).

The third choice, i.e., selecting which instructions to execute in compiled mode based on a profile, is only possible if a profile exists for the executable in question, as is determined by the name and the checksum of the executable. Note that every simulation run with SyntSim automatically generates such a profile. Since researchers tend to perform many simulations (with different parameters) on the same executable, a suitable profile should be available for all but the first run. In fact, even the first run can be performed quickly using SyntSim's heuristics. Alternatively, SyntSim can run the program in question with a short input to produce a profile for a following longer simulation run.

If a profile is present, the user can select all targets as compiled-mode entry points or specify a ratio. Since SyntSim profiles also contain the cumulative total number of instructions executed after each jump instruction during the profile run, SyntSim can select the top n targets such that the expected number of instructions executed in compiled mode over the expected number of instructions executed in interpreted mode is approximately equal to the user-selected ratio. Using a profile together with a ratio produces the most effective simulators because it

allows the user to minimize the overall time required for the synthesis, compilation, and simulation. For example, if the simulation time is expected to only be a few minutes, it does not make sense to spend tens of minutes to compile the simulator. Rather, in such a case a relatively low ratio should be specified, which will reduce the size of the synthesized C code and therefore the compilation time. Conversely, for long-running simulations a higher ratio should be specified because the longer compilation time can be amortized by the faster simulation speed.

3.2 Determining wanted code

Once the compiled mode entry points have been identified with any of the above methods, SyntSim continues the code discovery by determining the instructions that are reachable from these entry points. It visits all instructions sequentially and recursively follows branch targets until a jump, an illegal, or an invalid instruction is encountered. The visited instructions are marked for inclusion in compiled mode. The remaining instructions are deemed unwanted and will be handled by the interpreter if they need to be executed. No code is synthesized for the unwanted instructions.

3.3 Loading the translations

SyntSim includes a file that contains C code snippets (definitions) for all supported instruction. Currently, we have code for 194 instructions, including all instructions required to run the SPECcpu2000 programs with all provided inputs (test, train, and reference). It is straightforward to add code snippets for additional instructions. Each code snippet starts and ends with a special symbol. The start symbol includes an instruction number for identification purposes. Except for the delimiter symbols, the snippets comprise regular C code in which the reserved variables `A`, `B`, `C`, and `D` represent place holders for up to three source registers or immediates and a target register (see Section 3.4).

3.4 Simulator source-code synthesis

At this point, SyntSim is ready to generate the source code of the customized simulator. It starts out by emitting a fixed piece of startup code, which includes the system-call handler, memory access functions, emulators for three floating-point instructions with a special rounding mode, the interpreter, and predecode functions. The emulators for the three instructions are included because these instructions are rarely executed and require many lines of C code to simulate, making them bad candidates for inlining in compiled mode. The predecode functions determine the identification number and format for each instruction. The identification number is the same as the code-snippet number mentioned above. The eighteen possible instruction formats specify the number of sources and destinations as well as which sources are registers and which ones are immediates. The interpreter is a functional simulator that takes advantage of the pre-decode information. Moreover, it keeps track of the number of executed instructions and associates them with jump targets for later profile generation (see below). It transfers control to compiled mode whenever it encounters a jump. The emitted code also comprises the beginning of compiled mode, which is simply the first line of a switch statement used to jump to one of the compiled-mode code sections or the interpreter. This switch statement begins at line 5 in the following example, which shows true SyntSim output that has been reformatted for better readability. The comments were added by hand.

```
1:  static void RunCompiled()
2:  {
3:    ...  // local variable declarations
4:    while (1) {
5:      switch (pc/4) {
6:      case 0x4800372c:
7:        r[1] = r[29]+(8192<<16);        // 12000dcb0: ldah r1, 8192(r29)
8:        r[4] = r[29]+(8192<<16);        // 12000dcb4: ldah r4, 8192(r29)
9:        r[2] = RdMem8(r[1]-30768);      // 12000dcb8: ldq r2, -30768(r1)
10:       r[4] = r[4]-23232;              // 12000dcbc: lda r4, -23232(r4)
11:       r[0] = r[2]+1;                  // 12000dcc0: lda r0, 1(r2)
12:       r[2] = (r[2]<<3)+r[4];          // 12000dcc4: s8addq r2, r4, r2
13:       r[4] = 0+8192;                  // 12000dcc8: lda r4, 8192(r31)
14:       WrtMem8(r[0], r[1]-30768);      // 12000dccc: stq r0, -30768(r1)
15:       s1 = r[0];                      // 12000dcd0: cmplt r0, r4, r0
16:       s2 = r[4];
17:       r[0] = 0;
18:       if (s1 < s2) r[0] = 1;
19:       WrtMem8(r[16], r[2]+0);         // 12000dcd4: stq r16, 0(r2)
20:       r[16] = r[29]+(8191<<16);       // 12000dcd8: ldah r16, 8191(r29)
21:       ic += 12;
22:       if (0 != r[0]) goto L12000dcf0; // 12000dcdc: bne r0, 12000dcf0
23:       r[16] = r[16]-6456;             // 12000dce0: lda r16, -6456(r16)
24:                                       // 12000dce4: unop
25:                                       // 12000dce8: unop
26:       ic += 4;
27:       goto L12000c970;                // 12000dcec: br r31, 12000c970
28:       L12000dcf0:
29:       ic += 1;
30:       pc = r[26] & (~3ULL);           // 12000dcf0: ret r31, (r26), 1
31:       icnt[fnc(lasttarget)] += ic;
32:       ic = 0;
33:       lasttarget = pc;
34:       break;
35:     default:
36:       RunInterpreted();
37:     } // switch
38:   } // while
39:} // RunCompiled
```

Next, SyntSim sequentially emits translated code for each instruction that has been flagged for inclusion in compiled mode. No code is generated for NOPs (lines 24 and 25). Before each compiled-mode entry point (i.e., before each jump target), a case label is emitted (line 6) so that the switch statement can transfer control to it. The case "number" is the PC divided by four to reduce the size of the case-label table. Similarly, a normal label is

emitted before each branch target (line 28) so it can be reached using goto statements (line 22). A label's name is the target PC preceded by an "L". If the current instruction is the last instruction of a basic block, i.e., the instruction transfers control (line 22) or the next sequential machine instruction is a jump or branch target (line 30), a line of C code is emitted to increment the instruction counter by the number of instructions in the basic block (lines 21, 26, and 29). In case of a jump or system call, code is included to update the PC register (line 30). The PC is not needed in any other case and is therefore not updated elsewhere. At this point, SyntSim emits the code snippet for the current instruction. While doing so, it replaces all occurrences of the reserved variable `D` with the destination register of the instruction. The reserved variables `A`, `B`, and `C` are replaced with the first, second, and third source operand of the instruction, respectively. A source operand can either be a register (e.g., line 12) or an immediate from a literal or displacement field in the instruction word (e.g., line 11). For better performance, all uses of the hardwired zero registers are replaced by zeros (e.g., line 13).

In case of a jump instruction or a system call, SyntSim includes code to tally the number of executed instructions since the last jump for later inclusion in the profile (lines 31 through 33), assigns the new target to the PC register (line 30), and emits a break statement to leave (line 34) and subsequently loop back (line 38) to the switch statement. Then SyntSim proceeds with the next instruction until code for all wanted instructions has been generated. In the example above, we only show the synthesized code for one compiled-mode entry point (line 6). Typically, hundreds of entry points along with the code reachable from them are included in the switch statement.

Finally, SyntSim emits another fixed block of C code that includes the end of the switch statement (lines 35 onward), the function `main` (not shown), and the checksum of the original executable. The switch statement terminates with a default handler (line 35) that transfers control to the interpreter (line 36). Thus, whenever the simulator jumps to a target for which no compiled code exists, the interpreter is automatically invoked. Control is returned from the interpreter when the next jump instruction is executed. The `main` function contains code to set up and initialize the simulated stack, text (code), data, bss, and heap segments. The environment variables can optionally be included. Finally, `main` contains code to initialize the registers and to call the compiled-mode simulator with the program entry point in the PC register.

3.5 Compiling and running the simulator

Once the C code for the simulator has been generated, it is compiled. The resulting binary simulates the execution of the original program at high speed. To invoke the simulator, the user simply has to add the name of the binary before the normal command line used to launch the program, including any input and output redirections, parameters, etc. For example, instead of typing `myprogram myflags myinput`, one would simply enter `syntsim myprogram myflags myinput`.

4. Evaluation Methods

4.1 System

We performed all measurements presented in this paper on a 64-bit UP2000+ system with two 750MHz 21264A Alpha CPUs [23]. The second processor was not used. The CPUs have separate, 2-way associative, 64kB L1 caches and an off-chip, unified, direct-mapped 8MB L2 cache. The system is equipped with 2GB of main memory. The operating system is Tru64 UNIX V5.1.

4.2 Benchmark information

We used most of the programs from the SPECcpu2000 benchmark suite [21] for this study. We excluded perlbmk because it executes a fork, which is not yet supported. We also excluded eon because we were unable to finish the simulations in time for this paper. We further excluded the four Fortran 90 programs due to the lack of a compiler. For large amounts of compiled-mode code, the synthesized simulator for gcc cannot be compiled because the compiler runs out of memory. Hence, gcc results are only partially provided. To make all the averages in this paper consistent and comparable, we always exclude gcc when computing any mean, even for the cases where results are available.

Table 1: Information about the benchmark programs.

program	typ	lang	static	test	train	reference
gzip	integer	C	61,616	3,054.0 M	51,179.0 M	72,002.6 M
vpr		C	107,828	1,477.6 M	13,241.5 M	106,074.8 M
gcc		C	446,320	1,712.9 M	4,353.0 M	40,897.1 M
mcf		C	66,212	224.1 M	7,344.4 M	50,017.7 M
crafty		C	119,508	4,407.5 M	27,594.6 M	195,275.2 M
parser		C	108,744	3,318.8 M	11,371.8 M	464,206.5 M
gap		C	200,384	1,046.4 M	8,472.8 M	242,319.7 M
vortex		C	223,856	10,382.2 M	18,856.5 M	122,811.7 M
bzip2		C	59,456	7,536.1 M	50,809.5 M	102,713.3 M
twolf		C	132,044	251.1 M	12,670.6 M	342,984.1 M
mesa	floating point	C	213,776	2,506.1 M	68,019.2 M	279,515.7 M
art		C	68,076	2,862.5 M	7,218.9 M	76,878.9 M
equake		C	70,584	1,360.6 M	27,495.4 M	158,385.2 M
ammp		C	92,176	6,102.5 M	53,360.1 M	390,911.8 M
wupwise		F77	67,804	11,422.8 M	54,958.2 M	369,380.5 M
swim		F77	63,508	587.9 M	9,855.4 M	263,576.5 M
mgrid		F77	62,604	23,353.2 M	29,594.1 M	583,828.0 M
applu		F77	72,352	427.4 M	21,539.3 M	555,591.6 M
sixtrack		F77	303,328	14,269.7 M	172,680.2 M	809,506.4 M
apsi		F77	94,496	6,865.7 M	13,512.8 M	546,082.4 M
average			115,176	5,339.8 M	34,725.0 M	301,687.5 M

Note that averages in this paper refer to the arithmetic mean except for execution speed ratios, which we averaged using the geometric mean to de-emphasize outliers.

The C programs in the benchmark suite were compiled with Compaq's C compiler V6.3-025 using "-O3 -arch host -non_shared" plus feedback optimization. The C++ and Fortran 77 programs were compiled with g++/g77 V3.3 using "-O3 -static". SyntSim currently only supports statically linked Alpha binaries.

We run all programs to completion with the SPEC-provided test, train, and reference inputs. However, to save time, we limited the simulations to the first run for the programs for which multiple runs are specified.

Table 1 shows the program name, the type (integer or floating point), the programming language (C++, C, or Fortran 77), the number of static instructions in the executables, and the number of dynamically executed instructions (in millions) for the test, train, and reference inputs.

5. Results

Section 5.1 studies the speed of the synthesized simulators, Section 5.2 evaluates the impact of simulating caches and a branch predictor and provides a comparison with ATOM, Section 5.3 studies the effectiveness of SyntSim's built-in interpreter and compares it with SimpleScalar's *sim-fast*, Section 5.4 investigates compiler optimization levels, Section 5.5 discusses the performance of SyntSim's optimization features, and Section 5.6 takes a closer look at the synthesized C code.

5.1 Simulation speed

Figure 2 shows the speed of the simulators generated by SyntSim relative to native execution for each benchmark program when running the reference inputs (lower numbers are better). The results include the time to synthesize and compile the simulators. The left bars show the performance with a reference profile and a ratio of 1000:1, meaning that approximately 99.9% of the instructions are executed in compiled mode. The right bars show the performance with our heuristic-based approach, i.e., without the aid of profile information. No right bars are shown for gcc and sixtrack because the compiler runs out of memory when compiling those two simulators without a profile.

Our synthesized simulators are very fast. They execute all SPECcpu2000 programs with the reference inputs within a factor of 16 relative to native speed when a reference profile is available. The geometric mean slowdown is a mere 6.6. Surprisingly, in the best case (mcf), native execution is only twice as fast as the simulator. The slowdown on the integer programs is a little higher (7.3) than on the floating-point programs (6.0), which is expected because of the shorter basic blocks and the higher native IPC of the integer programs.

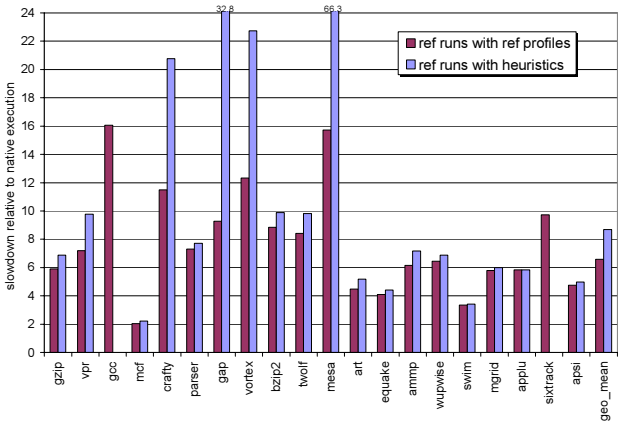

Figure 2: SyntSim's performance on the reference inputs.

Using heuristics instead of profile information results in somewhat slower simulators with a geometric mean slowdown of 8.7. Three programs, eon, gap, and mesa, are over a factor of two slower when using heuristics rather than profile information. In the other cases, our heuristics are very effective.

Figure 3 shows how much faster native execution is on average than SyntSim's simulators (lower numbers are better) for different program inputs, profiles, and user-selected ratios. Again, the results include the time to synthesize and compile the simulators.

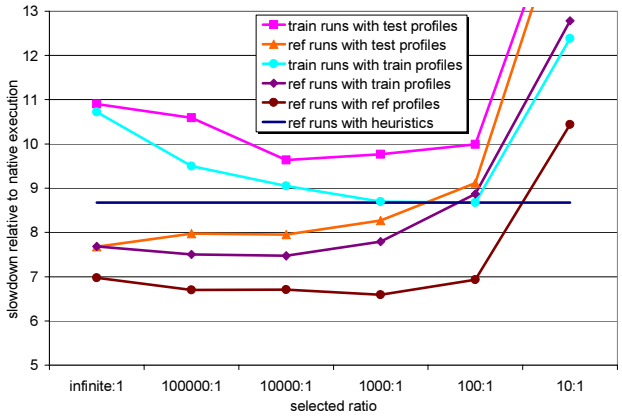

Figure 3: Performance with different ratios and profiles.

As Figure 3 shows, the simulators are only about seven to eleven times slower than native execution, regardless of the profile source and the ratio as long as the ratio is at least 100:1. The interpreter, on the other hand, incurs a slowdown of between fifty and sixty (Section 5.3) and is thus about eight times slower than mixed-mode execution.

SyntSim's performance on the reference inputs exceeds its performance on the train inputs. This is ex-

pected as the train runs are much shorter, making it harder to amortize the compilation overhead. Similarly, the performance on the train runs is better than on the test runs (results are off scale). SyntSim incurs a slowdown of 17.2 on the test inputs relative to native execution, which is still 3.7 times faster than interpretation, showing that out approach is also useful for short simulation runs. Note that we saw no case where mixed mode was slower than interpreted mode.

On the train runs with train profiles, using compiled mode exclusively is 24% slower than mixed mode with a ratio of 100:1, which is the best ratio for the these runs. For the reference runs, the best ratio is 1000:1 and for the train runs it is 100:1.

Interestingly, with imperfect profile information, the preferred ratios are higher. A ratio of 10,000:1 results in the fastest reference runs with profile information from the train inputs. With information from the test runs, it is best to include all code that was executed during the profile run (corresponding to an infinite ratio). The train runs prefer a ratio of 10,000:1 when only test input information is available. Evidently, the less accurate the profile information is, the higher a ratio should be chosen.

Figure 4 compares the performance of SyntSim's mixed mode with SyntSim's interpreter and SimpleScalar's *sim-fast* (lower numbers are better). *Sim-fast* does not execute all our binaries correctly, which is why some programs have been omitted from the figure. The mixed-mode results were obtained with a ratio of 1000:1 and a reference profile.

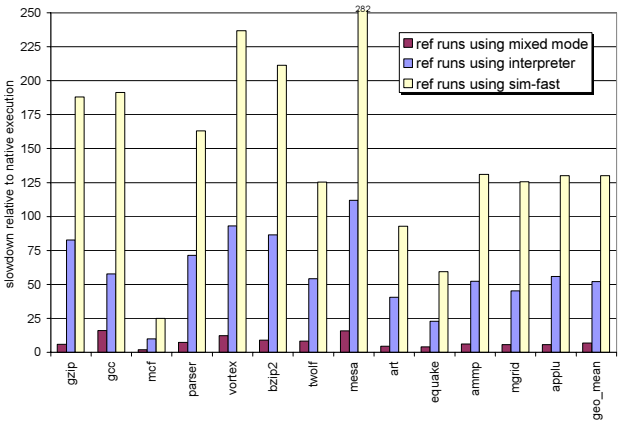

Figure 4: Performance of SyntSim and *sim-fast*.

SyntSim's mixed-mode simulation speed is 18.9 times faster than *sim-fast* on average (geometric mean). The performance benefit ranges from twelve times faster on gcc to 32 times faster on gzip. SyntSim's mixed mode outperforms its own interpreter roughly by a factor of eight, ranging from 3.6 times faster on gcc to fourteen times faster on gzip.

5.2 Performance of user-provided code

In this section, we evaluate the performance of our system when including user-provided C code and compare it with ATOM [11, 38] simulations that run the same code. We experimented with three pieces of code. The first one (ic) simply counts the number of executed instructions. The second one (ic+memh) also counts the dynamic instructions, but at the same time models a memory hierarchy with separate, two-way associative, 64kB first-level caches with a least-recently-used replacement policy and a unified, direct mapped, 4MB second-level cache. All three caches are write allocate and have 64-byte blocks. The third piece of user code is the same as the second one but also includes a 1024-entry gshare branch predictor (ic+memh+bp). We chose these experiments because counting instructions and warming up caches and branch-predictor tables is useful when fast-forwarding before switching to a cycle-accurate simulation mode.

Table 2 shows the average slowdown relative to native execution when running the three experiments on ATOM and SyntSim using the reference inputs (lower numbers are better).

Table 2: Performance of ATOM and SyntSim.

	slowdown with	
	ATOM	SyntSim
ic	3.4	6.6
ic+memh	30.2	11.5
ic+memh+bp	34.4	13.2

ATOM is almost twice as fast as SyntSim at counting the number of executed instructions. However, when we add the code for the memory hierarchy, SyntSim becomes over 2.6 times faster than the essentially identical ATOM code. The performance benefit remains about the same when further adding a branch predictor to the simulations. We believe that SyntSim is superior to ATOM in these cases because of its ability to inline the user code, which allows the code to be optimized in the context of the individual call sites and removes the call overhead.

Running the memory hierarchy and the branch predictor in addition to simulating and counting instructions slows down the SyntSim simulations by about a factor of two. In other words, SyntSim can perform these simulations at about one thirteenth of native speed.

5.3 Interpreter performance

In this section, we study the speed of SyntSim's interpreter (i.e., the performance when compiled mode is not used) relative to native execution. We also compare the interpreter's speed with that of *sim-fast*, the fastest functional simulators included with SimpleScalar [6].

Table 3 shows how much faster native execution is than interpreted execution for the three provided program inputs (higher numbers are worse).

Table 3: Interpretation time over native execution time.

program	test	train	ref
gzip	101.6	97.3	82.8
vpr	85.2	73.3	58.5
gcc	70.8	71.8	57.7
mcf	16.2	12.2	10.0
crafty	85.4	89.2	87.6
parser	77.6	73.7	71.4
gap	49.3	57.8	63.5
vortex	96.3	104.4	93.1
bzip2	105.5	105.2	86.6
twolf	72.1	61.1	54.1
mesa	105.4	110.7	111.9
art	37.3	38.7	40.6
equake	75.7	43.9	22.8
ammp	35.8	46.8	52.4
wupwise	66.6	62.6	62.3
swim	29.2	21.8	20.4
mgrid	44.8	69.8	45.3
applu	91.9	68.6	55.8
sixtrack	82.2	83.2	83.5
apsi	62.7	60.5	46.8
geo mean	62.9	60.2	53.0
maximum	105.5	110.7	111.9
minimum	16.2	12.2	10.0

While the average slowdown lies between about a factor of fifty to sixty, the individual slowdowns range from only ten to over 110 times. Surprisingly, mcf can be interpreted at one-tenth its native speed with the reference input. This is possible because mcf has by far the lowest native IPC (instructions per cycle) of only 0.14. swim's IPC of 0.32 is the second lowest and swim also incurs the second lowest slowdown when interpreted. mesa exhibits the highest IPC (1.69) and also incurs the highest slowdown when interpreted. However, for many of the remaining programs the native IPC does not correlate strongly with the interpreter's performance.

While the majority of the programs can be interpreted more efficiently with the longer inputs, there are several programs for which this is not true, most notably ammp. For mgrid, the train input incurs a much larger slowdown than the other two inputs.

On average, the integer programs experience a larger slowdown (58.8 to 68.6) than the floating-point programs (48.2 to 58.1). This may again be due to the native IPC, which is lower for floating-point programs.

Table 4 shows the interpreter performance of SyntSim relative to SimpleScalar's *sim-fast* (higher numbers are better). The baseline interpreter in SyntSim is faster than *sim-fast* on every program we tested. In fact, it is over 2.2 times faster even in the worst case and the performance advantage is quite consistent.

Table 4: Interpreter speed of SyntSim relative to *sim-fast*.

program	test	train	ref
gzip	2.44	2.37	2.27
gcc	2.56	2.76	3.31
mcf	3.07	2.52	2.51
parser	2.41	2.33	2.29
vortex	2.65	2.65	2.54
bzip2	2.62	2.58	2.44
twolf	2.38	2.32	2.31
mesa	2.58	2.46	2.52
art	2.30	2.32	2.28
equake	2.57	2.69	2.60
ammp	2.42	2.52	2.50
mgrid	2.78	2.80	2.77
applu	2.29	2.35	2.33
geo mean	2.53	2.49	2.44
maximum	3.07	2.80	3.31
minimum	2.29	2.32	2.27

5.4 Compiler optimization levels

Table 5 lists the average time in seconds (lower numbers are better) for the synthesis, the compilation, and the simulation when using different compiler optimization levels. The last column lists the total time, i.e., the sum of the three times in each row. The bolded entries mark the shortest total time for each input. The results for all inputs were obtained with perfect profile information and a ratio of 1000:1. Note that we used the "-arch host" flag in all cases to allow the compiler to take advantage of the full instruction set of the underlying CPU.

Table 5: Average synthesis, compilation, and simulation time with different optimization levels.

	level	synthesis	compilation	simulation	total
test	O0	0.09	23.56	102.79	126.43
	O1	0.09	44.92	53.05	**98.07**
	O2	0.09	63.20	55.34	118.63
	O3	0.09	63.42	55.37	118.88
	O4	0.09	66.02	56.31	122.43
train	O0	0.08	16.45	663.05	679.58
	O1	0.08	36.00	346.18	**382.26**
	O2	0.08	45.03	354.55	399.66
	O3	0.08	45.34	355.50	400.92
	O4	0.08	47.50	349.75	397.33
ref	O0	0.08	16.92	6386.44	6403.44
	O1	0.08	33.06	3160.25	**3193.38**
	O2	0.08	40.32	3337.93	3378.34
	O3	0.08	40.13	3332.42	3372.63
	O4	0.08	42.67	3325.71	3368.46

The synthesis time is independent of the optimization level and is insignificant since our algorithm's complexity is linear in the number of instructions. As we will see in Section 5.6, more instructions are synthesized on average for the shorter runs, which is why the corresponding synthesis time is slightly longer. The difference in the number of synthesized instructions does, however, have a considerable effect on the compilation time, which is longest

with the test and shortest with the reference profiles. Expectedly, the compilation time increases with the optimization level. It jumps substantially when going from O0 to O1 and from O1 to O2, but thereafter flattens out. This, together with the fact that the simulation time for the three highest optimization levels is quite similar, means that O3 and O4 do not optimize much more than O2. O3 adds inline expansion of global procedures, but SyntSim always synthesizes a single C file in which all functions are static (i.e., local). Hence, O3 should not add anything over O2. Indeed, the corresponding simulation times are within 0.3%, which is within the margin of error of our timing measurements. O4 adds software pipelining, vectorization, and more aggressive scheduling, which increases the compilation time a little bit. The resulting simulation time is worse than O2/O3 for the test inputs but better for the train and reference inputs. Nevertheless, O1 results in the fastest simulation speed. O2 adds optimizations that improve speed at the cost of extra code size (including inlining, loop unrolling, and code replication to eliminate branches). Apparently, this is a bad tradeoff in our already large code. After all, the average synthesized simulator (with train profiles and a ratio of 1000:1) comprises 53,700 lines of C code. We can only surmise that the code-increasing optimizations cause worse instruction-cache behavior and hence result in a slowdown rather than a speedup of the simulator. Overall, O1 is 21% faster than the other optimization levels for the test inputs, 4% for the train inputs, and 5% for the reference inputs. The difference on the test runs is much larger because the compilation time factors in more on shorter simulations.

Since O1 not only produces the fastest simulators but also results in the second lowest compilation time, it is the optimization level of choice. All other sections in this paper use O1 exclusively.

5.5 SyntSim's optimization effectiveness

In this section, we investigate how SyntSim's unwanted-code elimination, case-label minimization (to preserve basic blocks), and instruction counting (for profile purposes) impact the performance.

Table 6: Performance of some optimizations (train runs).

	compiled instrs	goto labels	case labels	synt. time	comp. time	simul. time	total time
no unwanted-code elim.	115176.4	7618.8	213.8	0.35	603.2	340.9	944.5
using heuristics	92278.1	6594.8	4707.3	0.31	489.6	373.1	863.1
case labels everywhere	20659.8	1701.4	20659.8	0.10	346.4	494.3	840.8
baseline	20659.8	1701.4	213.8	0.09	36.0	346.2	382.3
no profile generation	20659.8	1701.4	213.8	0.08	30.9	332.7	363.7

Table 6 shows the average number of synthesized instructions, goto labels, and case labels, as well as the average time in seconds to synthesize, compile, and run the simulators, and the total time. We list results for train runs with train profile information and a ratio of 1000:1 (where applicable), which represents the baseline.

The first row shows the effect of turning unwanted-code elimination off. The second row replicates the heuristic-based results from the previous section for comparison purposes. The third row lists numbers when emitting a case label before every synthesized instruction. The fourth row is the baseline case. The last row shows how much faster the baseline would be without the overhead of producing a profile.

Without unwanted-code elimination, SyntSim has to generate code for every instruction in the executable, including ones that will never be executed. Emitting all this code makes the compilation slow, taking on average over ten minutes per program. The actual simulation, however, is 1.5% faster than that of the baseline. We had expected it to be slower due to worse instruction-cache behavior. As it turns out, the compiler is also able to eliminate all unwanted code (judging from the size of the executables) since unwanted code cannot be reached because the necessary case labels are not included. In the process, the remaining code is probably moved around, resulting in a different code layout. We believe this to be the explanation for the 1.5% performance difference because looking at the results for each individual program, we find that some of them have a slightly shorter and some of them a slightly longer simulation time than their baseline counterparts. Overall, SyntSim's unwanted-code elimination is extremely fast and well worth it because it reduces the compilation time by a factor of 17.

By emitting case labels before every instruction, i.e., making every instruction a possible jump target, the concept of a basic block is lost, which severely limits the ability of the compiler to perform optimizations. In fact, the resulting code is 43% slower than the baseline. Together with the much longer compilation time, emitting case labels everywhere results in over twice longer turnaround times for the train runs.

Since SyntSim can make such good use of profile information, we felt it worthwhile to embed code to automatically generate profiles during every run, whether compiled, interpreted, or mixed mode is used. The last row in Table 6 shows that while doing so comes at a price, the overhead is not substantial. Emitting, compiling, and executing an increment statement for every basic block in compiled mode and every instruction in interpreted mode as well as some extra code before every indirect jump instruction adds 12% to the synthesis time and increases the compilation time by 17%, but only slows down the simulations by 4%. Overall, the cost of always generating profiles is a 5% slower turnaround time for the train runs (and even less for the reference runs), which we believe to be well worth the benefit SyntSim can derive from having profile information available for the next run.

5.6 Compiled-mode instructions

Table 7 shows how many of the 115,176.4 average static instructions in our benchmark programs are included in compiled mode. The table gives results for different ratios and information sources (test profile, train profile, reference profile, or heuristics). The table also lists the number of emitted goto and case labels.

Table 7: Number of emitted instructions and labels.

	selected ratio	instructions	labels	cases
test	inf:1	42,860.8	3,282.6	934.8
	100000:1	36,056.4	2,817.6	580.1
	10000:1	28,218.7	2,257.7	372.6
	1000:1	22,800.8	1,842.6	262.4
	100:1	17,362.3	1,428.6	147.9
	10:1	11,368.6	929.9	49.0
train	inf:1	43,166.7	3,310.6	944.9
	100000:1	30,547.9	2,422.5	436.8
	10000:1	26,137.7	2,107.5	315.1
	1000:1	20,659.8	1,701.4	213.8
	100:1	15,076.1	1,248.9	113.3
	10:1	10,089.7	824.3	36.7
reference	inf:1	43,250.9	3,307.4	946.4
	100000:1	27,548.9	2,204.1	333.5
	10000:1	23,413.9	1,893.3	236.7
	1000:1	18,222.9	1,472.5	148.4
	100:1	12,201.5	968.3	89.7
	10:1	8,615.7	662.4	32.2
heuristics		92,278.1	6,594.8	4,707.3

Lower ratios always result in fewer instructions and labels being emitted. Interestingly, the reference information results in the smallest number of instructions and labels for all ratios except infinity. Similarly, train information results in fewer instructions and labels than test information (except for an infinite ratio).

6. Summary and Conclusions

This paper presents and evaluates a technique to create portable high-speed functional simulators without access to source or assembly code. SyntSim, one possible implementation of our technique, automatically generates and compiles a simulator before every simulation that is highly tuned for the given executable. The synthesized simulators include special optimizations and make interleaved use of compiled and interpreted simulation. SyntSim only translates between 15% and 37% of the static instructions into compiled mode, which suffices to execute about 99.9% of the dynamic instructions in compiled mode. The slowdown incurred by interpreting the few remaining instructions is much lower than the time it would take to compile translations for them. In fact, our mixed-mode simulations are eight times faster than pure interpretation and 24% faster than pure compiled-mode execution. To further accelerate the simulations, SyntSim performs optimizations such as unwanted-code elimination and label minimization. We found that removing unwanted code (a form of dead code) in the synthesizer accelerates the compilation process by a factor of 17. The label minimization ensures that the concept of a basic block is retained, which allows the compiler to perform much more aggressive optimizations and results in a 43% simulation speedup. Finally, SyntSim includes code to automatically generate a profile during every run. This incurs a 5% overhead, but SyntSim generates 32% more effective simulators with profile information than when using heuristics to determine what code to include in compiled mode. The final result is a portable simulator in C source code that is on average only 6.6 times slower than native execution and 19 times faster than SimpleScalar's *sim-fast* when running the SPECcpu2000 programs with the reference inputs. SyntSim allows users to include additional code to extend the simulated ISA, to simulate new ISAs, or to simulate caches, branch predictors, etc. The user-provided code is automatically inlined and undergoes the same optimizations as the remaining code. For example, when simulating a memory hierarchy in this way, the simulators generated by SyntSim are 2.6 times faster than ATOM.

References

[1] A. Adl-Tabatabai, M. Cierniak, G. Lueh, V. M. Parikh and J. M. Stichnoth. "Fast, Effective Code Generation in a Just-In-Time Java Compiler." *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 280-290. May 1998.

[2] Alpha Architecture Handbook, Version 4. ftp://ftp.digital.com/pub/Digital/info/semiconductor/literature/alphaahb.pdf

[3] T. Austin, E. Larson and D. Ernst. "SimpleScalar: An Infrastructure for Computer System Modeling." *IEEE Computer*, Volume 35, Issue 2, pp. 59-67. February 2002.

[4] T. Ball and J. R. Larus. "Optimally Profiling and Tracing Programs." *ACM Transactions on Programming Languages and Systems*, Vol. 16:4, pp. 1319-1360. July 1994.

[5] C. Bechem, J. Combs, N. Utamaphethai, B. Black, R. D. S. Blanton and J. P. Shen. "An Integrated Functional Performance Simulator." *IEEE Micro*, pp. 26-35. May 1999.

[6] D. Burger and T. M. Austin. "The Simplescalar Tool Set Version 2.0." *Technical Report 1342*, *Computer Sciences Department, University of Wisconsin*. June 1997.

[7] C. Cifuentes and M. V. Emmerik. "UQBT: Adaptable Binary Translation at Low Cost." *IEEE Computer*, 33(3), pp. 60-66. March 2000.

[8] R.F. Cmelik and D. Keppel. "Shade: A Fast Instruction Set Simulator for Execution Profiling." *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pp. 128-137. May 1994

[9] T. M. Conte, M. A. Hirsch and K. N. Menezes. "Reducing State Loss for Effective Trace Sampling of Superscalar Processors." *International Conference on Computer Design*, pp. 468-477. October 1996.

[10] J. Emer, P. Ahuja, E. Borch, A. Klauser, L. Chi-Keung, S. Manne, S. S. Mukherjee, H. Patil, S. Wallace, N. Binkert, R. Espasa and T. Juan. "Asim: A Performance Model Framework." *IEEE Computer*, Volume: 35, Issue: 2, pp. 68-76. February 2002.

[11] A. Eustace and A. Srivastava. "ATOM: A Flexible Interface for Building High Performance Program Analysis Tools." *WRL Technical Note TN-44*, Digital Western Research Laboratory, Palo Alto. July 1994.

[12] R. M. Fujimoto and W. B. Campbell. "Direct Execution Models of Processor Behavior and Performance." *Winter Simulation Conference*, pp. 751-758. December 1987

[13] S. R. Goldschmidt and H. Davis. "Tango Introduction and Tutorial." Technical Report CSL-TR-90-410, Stanford University. 1990.

[14] M. Gschwind, E. R. Altman, S. Sathaye, P. Ledak and D. Appenzeller. "Dynamic and Transparent Binary Translation". *IEEE Computer*, 33(3), pp. 54-59. March 2000.

[15] J. W. Haskins and K. Skadron. "Minimal Subset Evaluation: Rapid Warm-Up for Simulated Hardware State." *International Conference on Computer Design*, pp. 32-39. September 2001.

[16] J. K. Hollingsworth, B. P. Miller and J. Cargille. "Dynamic Program Instrumentation for Scalable Performance Tools." *Scalable High-performance Computing Conference*, pp. 841-850. May 1994

[17] R. J. Hookway and M. A. Herdeg. "DIGITAL FX!32: Combining Emulation and Binary Translation." *Digital Technical Journal*, 9(1). August 1997.

[18] http://h30097.www3.hp.com/dtk/spike_ov.html

[19] http://www.denkart.com/astoc/index.htm

[20] http://www.pennington.com/xtran.htm

[21] http://www.spec.org/osg/cpu2000/

[22] C.J. Hughes, V.S. Pai, P. Ranganathan and S.V. Adve. "Rsim: Simulating Shared-Memory Multiprocessors with ILP Processors." *IEEE Computer*, Volume 35, Issue 2, pp. 40-49. February 2002.

[23] R. E. Kessler, E. J. McLellan and D. A. Webb. "The Alpha 21264 Microprocessor Architecture." *International Conference on Computer Design*, pp. 90-95. October 1998.

[24] V. Krishnan and J. Torrellas. "A Direct-Execution Framework for Fast and Accurate Simulation of Superscalar Processors." *International Conference on Parallel Architectures and Compilation Techniques*, pp. 286-293. October 1998.

[25] T. Lafage and A. Seznec. "Choosing Representative Slices of Program Execution for Microarchitecture Simulations: A Preliminary Application to the Data Stream." *Kluwer International Series in Engineering and Computer Science Series, Workload Characterization of Emerging Computer Applications*, pp. 145-163. 2001.

[26] S. Laha, J. H. Patel and R. K. Iyer. "Accurate Low-Cost Methods for Performance Evaluation of Cache Memory Systems." *IEEE Transactions on Computers*, Volume C-37(11), pp. 1325-1336. February 1988.

[27] E. Larson, S. Chatterjee and T. Austin. "MASE: A Novel Infrastructure for Detailed Microarchitectural Modeling." *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 1-9. November 2001.

[28] J. R. Larus and E. Schnarr. "EEL: Machine-Independent Executable Editing." *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 291-300. June 1995.

[29] G. Lauterbach. "Accelerating Architectural Simulation by Parallel Execution of Trace Samples." *Hawaii International Conference on System Sciences*, Volume 1: Architecture, pp. 205-210. January 1994.

[30] P. S. Magnusson. "Efficient Instruction Cache Simulation and Execution Profiling with a Threaded-Code Interpreter." *Winter Simulation Conference*, pp. 1093-1100. Dec. 1997.

[31] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt and B. Werner. "SimICS: A Full System Simulation Platform." *IEEE Computer*, Volume 35:2, pp. 50-58. February 2002.

[32] H. Massalin. "Synthesis: An Efficient Implementation of Fundamental Operating System Services." *PhD thesis, Columbia University*. 1992.

[33] A. Nguyen, M. Michael, A. Sharma and J. Torrellas. "The Augmint Multiprocessor Simulation Toolkit for Intel x86 Architectures." *International Conference on Computer Design*, pp. 486-490. 1996.

[34] A. Paithankar. "AINT: A Tool for Simulation of Shared-Memory Multiprocessors." *Master's Thesis*, *University of Colorado at Boulder*. 1996.

[35] E. Perelman, G. Hamerly and B. Calder. "Picking Statistically Valid and Early Simulation Points." *International Conference on Parallel Architectures and Compilation Techniques*, pp. 244-255. September 2003.

[36] M. Rosenblum, E. Bugnion, S. Devine and S. A. Herrod. "Using the SimOS Machine Simulator to Study Complex Computer Systems." *ACM Transactions on Modeling and Computer Simulation*, pp. 78-103. January 1997.

[37] T. Sherwood, E. Perelman, G. Hamerly and B. Calder. "Automatically Characterizing Large Scale Program Behavior." *International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 45-57. October 2002.

[38] A. Srivastava and A. Eustace. "ATOM: A System for Building Customized Program Analysis Tools." *Conference on Programming Language Design and Implementation*, pp. 196-205. June 1994.

[39] J. E. Veenstra and R. J. Fowler. "MINT: A Front End for Efficient Simulation of Shared-Memory Multiprocessors." *Second International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pp. 201-207. January 1994.

[40] E. Witchel and M. Rosenblum. "Embra: Fast and Flexible Machine Simulation." *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pp. 68-79. May 1996.

[41] R. E. Wunderlich, T. F. Wenisch, B. Falsafi and J. C. Hoe. "SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling." *International Symposium on Computer Architecture*, pp. 84-97. June 2003.

[42] B. Yang, S. Moon, S. Park, J. Lee, S. Lee, J. Park, Y. C. Chung, S. Kim, K. Ebcioglu and E. Altman. "LaTTe: A Java VM Just-in-Time Compiler with Fast and Efficient Register Allocation." *International Conference on Parallel Architectures and Compilation Techniques*. October 1999.

[43] C. Zheng and C. Thompson. "PA-RISC to IA-64: Transparent Execution, No Recompilation." *IEEE Computer*, 33(3), pp. 47-52. March 2000.

[44] V. Zivojnovic and H. Meyr. "Compiled HW/SW co-simulation." *33rd ACM IEEE Design Automation Conference*, pp. 690-695. June 1996.