Continuous Optimization

Brian Fahs Todd Rafacz Sanjay J. Patel Steven S. Lumetta
Center for Reliable and High Performance Computing
University of Illinois at Urbana-Champaign
{*bfahs,trafacz,sjp,steve*}*@crhc.uiuc.edu*

Abstract

This paper presents a hardware-based dynamic optimizer that continuously optimizes an application's instruction stream. In continuous optimization, dataflow optimizations are performed using simple, table-based hardware placed in the rename stage of the processor pipeline. The continuous optimizer reduces dataflow height by performing constant propagation, reassociation, redundant load elimination, store forwarding, and silent store removal. To enhance the impact of the optimizations, the optimizer integrates values generated by the execution units back into the optimization process. Continuous optimization allows instructions with input values known at optimization time to be executed in the optimizer, leaving less work for the out-of-order portion of the pipeline. Continuous optimization can detect branch mispredictions earlier and thus reduce the misprediction penalty. In this paper, we present a detailed description of a hardware optimizer and evaluate it in the context of a contemporary microarchitecture running current workloads. Our analysis of SPECint, SPECfp, and mediabench workloads reveals that a hardware optimizer can directly execute 33% of instructions, resolve 29% of mispredicted branches, and generate addresses for 76% of memory operations. These positive effects combine to provide speed ups in the range 0.99 to 1.27.

1. Introduction

Dynamic optimization offers opportunities beyond static compiler optimization because of its ability to dynamically identify hot execution paths and adapt to changes in program behavior. Many dynamic optimizers [1, 2, 5, 7, 8, 9, 11, 20] share a common overall structure: they (1) select regions (functions, traces, hyperblocks, etc.) through some form of dynamic profiling, (2) optimize the selected regions, (3) cache the optimized versions, and (4) exchange future occurrences of the original regions with the optimized versions. We propose a dynamic optimization system, which we call *continuous optimization*, that does not require profiling of the instruction stream or caching of the optimized instructions. Instead, dataflow optimizations are applied to each fetched instruction using a table-based hardware optimizer located directly in the processor pipeline.

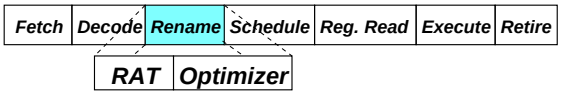

Figure 1. High-level view of optimizer integrated into a dynamically-scheduled processor pipeline.

As depicted in Figure 1, continuous optimization is integrated with the register alias table (RAT) in the rename stage of a dynamically-scheduled processor. The continuous optimizer uses the hardware tables described in Sections 2 and 3 to perform constant propagation, reassociation, redundant load elimination, store forwarding, and silent store removal. Execution results are fed back to, and exploited by, the optimizer to enhance performance. This feedback combined with optimization allows 33% of retired instructions to be non-speculatively executed early in the pipeline by the optimizer. The optimizer's objectives are (1) to reduce dataflow height, and (2) to execute simple instructions during optimization. These objectives are symbiotic, and combine to provide an average speed up of 1.11 over a standard pipeline.

Some previously proposed hardware-based dynamic optimizers [1, 9] perform classical compiler optimizations offline using an abstract optimizer operating on discrete instruction traces. Although continuous optimization can be adapted for optimizing instruction traces, it is not limited to discrete regions, i.e., it can impact a greater span of instructions. There have been numerous in-pipeline optimization techniques [4, 6, 21, 23, 24, 27]. Continuous optimization subsumes and extends many of them by aggressively optimizing dataflow through registers and memory to reduce dataflow height and to increase instruction-level parallelism (ILP).

This paper makes several contributions. First, the notion of continuous optimization is presented along with a

detailed description of a hardware implementation of such an optimizer. Second, continuous optimization's impact and sensitivity to design choices are characterized. Third, continuous optimization's contributing performance factors are analyzed. Finally, a quantitative comparison is made with other in-pipeline optimization techniques, and continuous optimization is shown to provide significantly higher performance.

This paper is organized as follows. Section 2 provides motivation and high-level details for continuous optimization. Hardware requirements are discussed in Section 3. Section 4 presents our experimental infrastructure. Section 5 provides the performance characterization, and Section 6 presents sensitivity studies. Section 7 presents related work along with comparisons to previously proposed in-pipeline optimization techniques. Section 8 provides a summary of the findings.

2. Continuous optimization

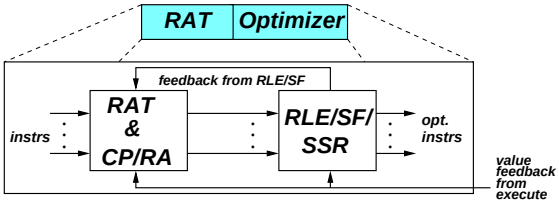

Figure 2. Architectural view of optimizer.

Figure 2 provides more details of the optimizer. Constant propagation (CP) and reassociation (RA) are implemented with additional logic integrated into the register alias table (RAT). Redundant load elimination (RLE), store forwarding (SF), and silent store removal (SSR) are performed with a separate, cache-like structure accessed after the RAT. Following their decoding, all instructions have their inputs and outputs renamed while being simultaneously transformed to a more parallel form. The crux of the optimizer is a *symbolic* representation of each architectural register value, which is maintained in the optimization tables. The symbolic representation is leveraged to increase dataflow parallelism and to reduce memory accesses. As an enhancement, execution results are fed back to the optimizer to increase effectiveness. We call this process *value feedback*, and call the values fed back *known values* to differentiate them from symbolic information available before instructions execute. Instruction optimization does not stall waiting for value feedback because symbolic values suffice for correctness. Bekerman et al [4] was first to propose value feedback for early load address resolution.

We now discuss the operation of the optimizer, explain and motivate its design, and discuss positive and negative implications of continuous optimization.

2.1. Symbolic register values

All optimizations operate on *symbolic* register values of the form `(reg<<scale)±offset`, where `reg` is a physical register, `scale` is a two-bit left shift quantity[1], and `offset` is a 64-bit immediate field. The symbolic expression was chosen because it enables a variety of optimizations, as described in the next section, and approximately 55% of instructions executed match its form[2].

2.2. Optimizations

The optimizations performed are described below:

Constant propagation (CP) propagates known values from producers to consumers. Simple instructions with known constant inputs can be executed in the optimizer. If, for example, `add r3, 4 -> r4` is being optimized, and `r3` is known to be `3`, the add is performed and `7` is moved into `r4`. This optimization subsumes constant folding.

Reassociation (RA) transitively flattens symbolic expressions, reducing dataflow height and increasing ILP by transferring dependences to earlier producers. A consumer's input is copied from its producer's input, and its `offset` and `scale` are recalculated. This optimization subsumes copy propagation.

Redundant load elimination (RLE) combines load operations accessing identical memory locations; the loads after the first are converted to move operations, which are then optimized away in a manner similar to [11, 15].

Store forwarding (SF) converts loads referencing recently stored values into move operations, which are then optimized away as in redundant load elimination.

Silent store removal (SSR) removes store operations that write a value identical to the currently stored value.

RLE/SF/SSR follow CP/RA because constant propagation and reassociation simplify `reg±offset` address specifications, enabling effective redundant load elimination, store forwarding, and silent store removal. Multiple optimization passes are not performed, but the benefits of doing so are achieved by feeding redundant and store forwarded load information back to the CP/RA stage; CP/RA then copy propagates the information through subsequent instructions.

Our memory optimizations are not speculative. Therefore, removed load and store instructions are safely eliminated with respect to a single thread of execution. We assume that memory-mapped I/O is identifiable. In a real system, where another thread of execution may disturb a memory location unbeknownst to the thread being optimized,

1 Left shifts of 0, 2, and 3 corresponding to addx, s4addx, and s8addx of the Alpha ISA are the only allowed shift values.

2 Removing `scale` decreases coverage to 54%. Expressions can be 32-bit or 64-bit; without 32-bit support, coverage decreases to 50%.

such optimizations may not be possible. We evaluate the impact of performing strictly safe optimization in Section 5.2.

2.3. Performing other optimizations

Low-level compiler optimizations can be divided into two categories[3]: *backward* and *forward* optimizations. Backward optimizations, such as dead code elimination, are efficiently performed by analyzing instructions in reverse sequential order. We do not remove dead code for two reasons. First, when performed non-speculatively, dead code removal requires instruction buffering to identify consumers and anti-dependences. Since continuous optimization is timing critical and buffering instructions impacts timing, dead code elimination is probably not a good design choice. Additionally, dead code elimination's primary benefit, from our experience, is reduction of fetch bandwidth requirements. Since instructions have already been fetched, much of the benefit of removing them is lost. We believe dead code should be removed in an off-the-critical-path dynamic optimizer, such as in [9], where latency is less critical. A speculative, in-pipeline technique for dead code removal [6] was shown to improve performance on a resource-bound machine. Including this technique, however, more than doubles our required chip real estate, but is a plausible addition for a resource-bound machine.

Forward optimizations operate on instructions in sequential order. They can be subdivided into *disconnected* and *connected* dataflow optimizations. Disconnected dataflow optimizations identify and connect similar operations that are not linked through dataflow. Connected dataflow optimizations reduce dataflow height, increase ILP, and increase dead code by cutting dataflow dependences. Constant propagation, reassociation, and store forwarding are examples of connected dataflow optimizations. We include the disconnected dataflow optimizations, redundant load elimination and silent store removal, because they are simple extensions to our store forwarding optimization. We could have also included general instruction reuse [24], which is an in-pipeline technique for optimizing common subexpressions. Adding it could enhance continuous optimization, and continuous optimization canonicalizes instructions, potentially making instruction reuse more powerful. Their combination is the subject of future work.

Continuous optimization subsumes and extends some previous in-pipeline optimization approaches. Reverse integration [24] uses a clever trick to extend general instruction reuse to optimize limited instances of store forwarding and reassociation, i.e., mostly stack references and stack pointer updates. Early load address resolution [4] uses a limited form of constant propagation and reassociation to pre-compute memory addresses to issue loads earlier in the pipeline. Load and store reuse [23] and speculative memory bypassing [21] are alternative approaches to performing redundant load elimination, store forwarding, and silent store removal in the pipeline. Continuous optimization extends these approaches by aggressively optimizing all dataflow, i.e., through registers and memory, to reduce dataflow height and increase ILP. Additionally, continuous optimization pre-executes 29% of mispredicted branches, which reduces misprediction penalty. To our knowledge, this important optimization is unique. In Section 7.1, we compare continuous optimization with reverse integration, speculative memory bypassing, and early load address resolution and demonstrate that continuous optimization's additional benefits provide significant performance improvements over these prior works.

2.4. Value feedback

Integrating execution results into the optimization tables allows symbolic values to be converted into *known values*, which can then be propagated as constants to consumers. For example, if `add r3, 4 -> r4` generates the result `15` in physical register `pr22`, and `pr22` is still referenced in the optimizer tables, expressions referencing `pr22` are updated with the value `15`. Subsequent instructions using expressions containing `pr22` use the value `15` for `pr22` and potentially execute during optimization. The latency required for a symbolic expression to become a known value depends on the pipeline length between the optimization and execution stages and the time required to transmit the result back to the optimization stage once the instruction producing the result has executed.

2.5. Motivating example

To demonstrate the opportunities for continuous optimization, we use the code example in Figure 3. *Static Code* is the static representation of a loop that sums the elements of an array. Assume that the loop counter is initialized to some value that is not statically computable. On each iteration, an array element is loaded and added to the sum, the loop counter is decremented, and the next array index is computed. The loop ends when the loop counter reaches zero. *Dynamic Data Flow* illustrates producer-consumer relationships for the loop instructions as they execute. Each array index addition and array element load within each iteration are fed by a loop-carried dependence. Similarly, each iteration's branch is dependent on a chain of subtractions equal in length to the chain of array index additions.

The instruction sequence emitted by the continuous optimizer is labeled *Optimized Data Flow*. The arcs modified by optimization are shaded. Although the dataflow height of

3 This categorization applies for basic block optimization, trace-based optimization, and optimization on the dynamic instruction stream like that performed here. It may not be directly applicable to general static compiler optimizations.

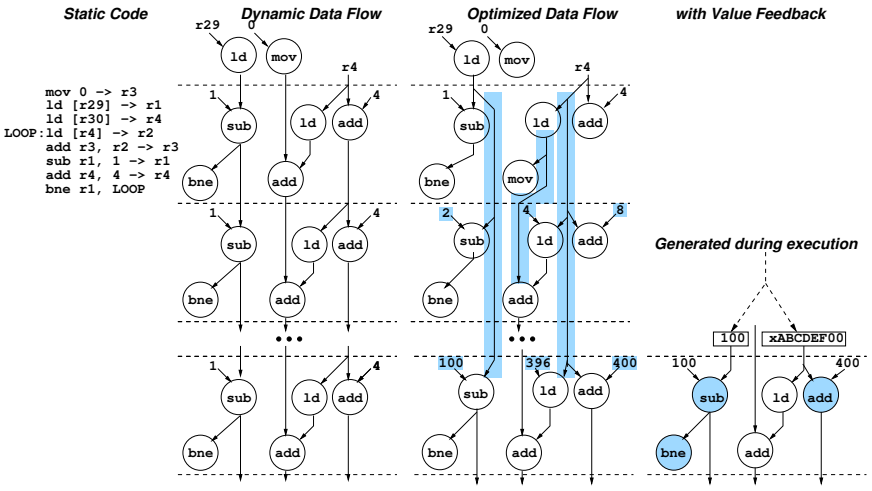

Figure 3. Motivating example.

the accumulate chain is not significantly reduced, the computation chains for the array index, load, and loop counter have been eliminated.

Consider what happens when `sub r1, 1 -> r1` from the first iteration enters the optimization stage. The CP/RA tables are accessed by the source architectural register number `r1`, and the previous mapping of `r1` is discovered (previously generated by `ld [r29] -> r1`). For example, assume `r1` was previously mapped to physical register `pr35`. Because the `sub` also writes `r1`, `r1`'s mapping is updated with the symbolic value `pr35-1`. When `sub r1, 1 -> r1` from the second iteration is optimized, `r1` maps to `pr35-1`, and the optimizer replaces this with `pr35-2`. If the physical register destination for the second `sub` is `pr37`, it becomes `sub pr35, 2 -> pr37`.

By the 100^{th} iteration, instructions in the loop's prologue have probably completed execution, in which case their results are in the optimization tables (value feedback). The column labeled *with Value Feedback* shows changes in the 100^{th} iteration that result from incorporating execution results into optimization. In particular, both the iteration counter load (`ld [r29] -> r1`) and the array base load (`ld [r30] -> r4`) have their results integrated into the optimization tables because both are still live. The shaded instructions can be executed during optimization because their inputs are constant. The out-of-order portion of the pipeline does not need to execute them. Additionally, the memory address for the load can be computed, which allows it to bypass the out-of-order portion of the pipeline and immediately become eligible to access the cache.

This example does not use redundant load elimination, store forwarding or silent store removal, but a similar process applies to these optimizations.

2.6. Impact of continuous optimization

For new microarchitectural features, both positive and negative aspects must be considered. We now discuss continuous optimization's potential impacts.

2.6.1. Positives. Continuous optimization can improve performance or power or both.

Dataflow height reduction is provided by most optimizations and is beneficial because it increases ILP.

Early execution, i.e., executing instructions during optimization, has multiple advantages. First, it creates a synergistic effect: execution results are propagated to consumers, which might also execute early. Additionally, it relieves pressure on the out-of-order part of the pipeline because instructions executed early only need to retire, i.e., they do not pass through the scheduler, dispatch, register read, and execute stages. For the benchmarks we evaluate, 33% of retired instructions execute early, and 29% of mispredicted branches resolve during optimization. A recent Pentium 4 [13] has a minimum branch misprediction penalty of over 30 cycles (the majority occur after rename). Almost all post-rename cycles can be saved when a mispredicted branch executes early.

Load and store reduction is provided by redundant load elimination, store forwarding, and silent store removal. These optimizations exploit provable short-term data reuse to remove an average of 21% of load and 2% of store instructions, potentially reducing power because optimizer table reads likely consume less power than cache accesses.

2.6.2. Negatives. The positive aspects come at some cost. We now list potentially negative aspects of continuous optimization.

Increased pipeline depth is a potential drawback, however, the number of stages required for optimization is likely to be small, on the order of two to four. Adding pipeline stages increases misprediction penalty, and, con-

versely, early branch resolution reduces the penalty for branches resolved during optimization. These effects counteract each other; the resulting impact depends on the number of branches recovered early.

Design complexity potentially increases. Fast, simple ALUs are required for each instruction that can pass through rename in a single cycle (e.g., four in a four-wide rename stage), plus bypass logic for optimization. Additionally, value feedback carries execution results back to the rename stage, which requires forwarding logic absent in current processors. It might appear as if more physical registers are required with continuous optimization. However, because of value feedback and early execution, physical register pressure almost always decreases, often significantly (physical register pressure increased for 2 of 23 benchmarks, g721 decode and g721 encode). The optimization tables require approximately 2K bytes of storage plus storage for recovering optimization state on branch mispredictions. The CP/RA table requires a 100–150 bit entry per architectural register and read and write ports equivalent to the RAT. The RLE/SF/SSR stage is a small cache, modeled with 32 entries of 100–150 bits; it requires read and write ports for each load and store capable of being optimized each cycle. The complexity described here is the worst case. Since optimization improves performance, but is not necessary for correctness, design complexity is an implementation variable, i.e., there is a tradeoff between performance and design complexity.

Power implications for the additional pipeline stages [12] and increased rename complexity are unclear. Certainly, without simplifying the out-of-order portion of the pipeline, power increases; however, optimizations simplify and pre-compute instructions. Thus, opportunity exists to scale back the design of the out-of-order core, potentially reducing dynamic power. We leave this subject for future work.

3. Microarchitectural details

The optimization process consists of two sequential steps. CP/RA are performed first and concurrently with register renaming; RLE/SF/SSR are second. Figure 4 shows the logic slice needed to process one instruction in a multi-instruction rename bundle.

3.1. CP/RA

Constant propagation and reassociation require symbolic expressions per architectural register. Upon entering rename, instructions' input source operands access the RAT and read renamed register mappings along with symbolic expressions. The optimizer processes the symbolic sources and either (1) executes the instruction (early execution), (2) derives a new symbolic form of the output (optimization), or (3) rules the output's symbolic form to be unexpressable (no optimization). The result is stored in the RAT for future instructions. Implementation-wise, constant propagation and reassociation are equivalent transformations. For constants, `reg` of the symbolic expression (`base phys. reg.` in Figure 4) is a hard-wired zero register, and a 64-bit constant is stored in the `base reg. val.` field.

The symbolic inputs are passed to an ALU, which executes the instruction if all values are known. Otherwise, depending on opcode, a new offset or physical register input(s) or both may be produced for the instruction. Nothing is done if the instruction's result cannot be encoded symbolically. The destination's RAT entry is updated with the instruction's physical register output and symbolic representation.

Careful consideration reveals that processing multiple instructions per cycle may be problematic. An instruction's optimized symbolic form may be required as input for an instruction in the same rename bundle. For a four-wide renamer, the implication is that four serial additions are required. For example, assume the following instruction sequence is a single rename bundle.

```
add  r1, 1 -> r2
add  r2, 1 -> r3
add  r3, 1 -> r4
add  r4, 1 -> r5
```

This sequence can be converted into four parallel instructions, but only through multiple serial additions. In the design we evaluate in Section 5, we only optimized the first addition in a chain of additions. Section 6.1 examines the implications of this choice.

A second subtlety arises regarding physical register lifetime. Physical register deallocation schemes, such as those in the MIPS R10000 and Alpha 21264, deallocate physical registers after retirement of the next instruction that overwrites the architectural destination. Optimizations can extend physical register lifetime beyond this point. Therefore, we instead use a reference counter algorithm like that proposed by [15].

3.2. RLE/SF/SSR

Redundant load elimination, store forwarding, and silent store removal transform loads and remove unnecessary stores. Fundamentally, this stage works precisely like CP/RA, except that instead of indexing with architectural register inputs, it is indexed with the data address for the memory operation. Load and store instructions check the table for previous memory operations that accessed the same memory location. At rename, memory addresses are generally unknown, however, CP/RA increases known data addresses to 76%. If a load has a known data address, it is looked up in a small table called the Memory Bypass Cache (MBC). A hit provides the symbolic

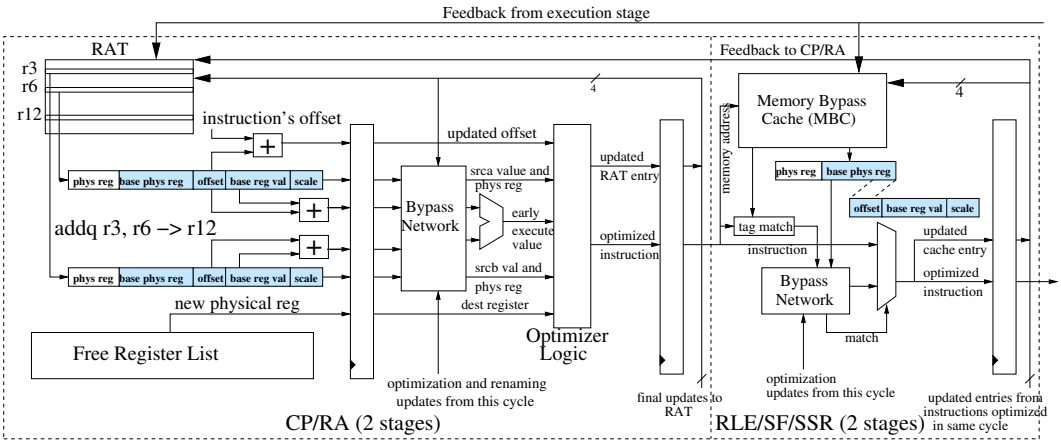

Figure 4. Microarchitecture of the optimizer showing the logic slice to optimize one instruction.

representation of the data for the load (no data cache access is required). If the load's data address misses the MBC, a symbolic expression of the load's destination is inserted into the table (for redundant load elimination). If the load's data address is unknown (i.e., not computed by CP/RA), no lookup is performed and nothing happens.

If a store with a known address hits in the MBC and the symbolic expression matches the store's symbolic expression of the data (propagated from CP/RA), the store is eliminated (for silent store removal). If the symbolic data does not match or the store's data address misses, then a symbolic representation of the store data (propagated from CP/RA) is inserted (for store forwarding). If an unknown store address is encountered, no lookup is performed, and the MBC is flushed.

Excluding MBC tag information, the MBC and CP/RA entries are identical. For simplicity, MBC addresses are 8-byte aligned. Tag matching compares the standard address tag, the offset from the 8-byte alignment, and access size. If a load hits, the data (symbolic expression) is forwarded to all intermediate references (instructions in the current and previous pipeline stages), written back to the CP/RA table updating the load destination information, and used to convert the load into the symbolic expression. As with sequential additions for CP/RA, dependences across instructions within a rename packet are not satisfied with RLE/SF/SSR either. We use a 32 entry MBC by default. We experimented with MBC sizes varying from 32 to 256 entries and found no difference in average performance. We also experimented with variable alignments (i.e., besides 8-byte alignments), but found that performance only improved for a single application, untoast.

Rather than flushing the MBC on unknown store addresses, we could proceed speculatively. Additionally, we could track unknown memory addresses by hashing `reg`$\pm$`offset` address expressions. We evaluated both extensions and found that misspeculations overshadow the improvements in memory bypassing. We have not yet evaluated predictive techniques that could reduce these misspeculations. We leave this analysis for future work. For brevity, we omit our detailed analysis on various MBC configurations from this paper.

3.3. Miscellaneous issues

In this section, we discuss some details that complicate the implementation for the proposed hardware optimizer. They are abstracted away during evaluation, but a real implementation must consider them, as they are necessary for the optimization hardware.

Store forwarding and redundant load elimination forward symbolic expressions through memory. For effectiveness, the newly found dataflow must be copy propagated. To avoid multiple optimization iterations while achieving its benefits, forwarded symbolic expressions are transmitted back to the CP/RA stage. Doing so in a straightforward manner adds write ports to the RAT (CP/RA table), a potentially objectionable requirement. However, the CP/RA table can be split into two structures. One structure, the RAT, is updated by instructions currently being renamed (as it is normally). The second structure, the Copy Propagation Table (CPT), is updated by feedback from the RLE/SF/SSR stage. When instructions enter rename, they read their input registers' symbolic expressions from both tables in parallel. For each input, a MUX determines the symbolic expression to use. When an instruction updates the RAT with its physical register and symbolic expression, it also invalidates the corresponding entry in the CPT. Splitting the CP/RA table avoids the additional write ports to the RAT at the expense of complexity and additional delay in evaluating instructions.

Another implementation obstacle to consider is value feedback. Execution results are forwarded back to the optimizer to enhance optimization. However, physical register value updates are problematic because a physical register may be referenced multiple times in the optimiza-

tion tables. To update multiple locations simultaneously with the same value, either a content-addressable structure or a level of indirection is required. When a physical register value arrives, each entry can perform a content-addressable match with its `base phys. reg`. If it matches, the `base phys. reg.` is set to the zero register and the `base reg. val.` is set to the incoming value. With the indirection approach, the `base phys. reg.` can be examined in a separate value table, but this adds extra latency to the optimizer, potentially complicating the intra-optimizer bypass network.

3.4. Continuous vs. discrete optimization

Although the optimizations have been described for continuous optimization, the hardware can be adapted for offline hardware-based optimizers, such as rePLay [9], PARROT [1], and trace-cache-based schemes [11, 14]. The basic structure remains identical. The difference between online and offline optimization is that optimization tables are invalidated at the start of each trace (or frame); offline optimizations are discrete per region as opposed to continuous across the execution stream. Furthermore, real-time value feedback for discrete optimization does not occur. On the other hand, multipass, reverse pass (dead code removal), and complex optimizations are easier offline.

4. Experimental setup

In this section, we describe the benchmark suites we use for evaluation and our default processor model.

4.1. Experimental workload

We use the SPEC2000 integer, SPEC2000 floating point, and mediabench benchmarks in Table 1. We provide results for all benchmarks working in our infrastructure. For SPECint, we modified the input sets to allow simulation through program completion, and we believe our modifications preserve original input set characteristics. The C and C++ benchmarks were compiled at optimization level -O4 with the Compaq Alpha C V5.9 and Compaq Alpha C++ V6.5 compilers (bzip2 used -O3 because -O4 produced an incorrect program). At level -O4, both compilers perform loop unrolling, software pipelining, vectorization, inline expansion, and other optimizations. The Fortran benchmarks were compiled with the HP Alpha Fortran V5.5A compiler with optimization level -O5. At this level, the compiler performs the same optimizations as the C and C++ compilers and also adds loop transformation optimizations.

4.2. Performance model

Our default processor model, outlined in Table 2, resembles the Pentium 4 [13] and is built using the SimpleScalar 3.0 framework. We believe that CP/RA can be overlapped with rename, which is modeled with two pipeline stages. Therefore, when continuous optimization is simulated, only two pipeline stages are added to rename for RLE/SF/SSR, which adds two cycles to the branch misprediction penalty for branches not resolved during optimization. For mispredicted branches resolved early, recovery happens after the extended rename stage. Execution results being fed back to the optimizer require one cycle for transmission. Only one level of addition dependence is evaluated in a cycle. If one addition feeds another addition within a rename bundle, the dependent is not optimized. Similarly, if the result of one load is used for the address of another within a rename bundle, the dependent is not optimized. Sensitivity to this configuration is examined in Section 6. For all optimizations, correctness is ensured through expression and value checking to avoid faulty optimizations.

5. Performance characterization

In this section, we evaluate several performance aspects of continuous optimization.

5.1. Speedup over the baseline

Figure 5 demonstrates speed up of continuous optimization compared to the baseline processor. Average speed up is the rightmost bar. Despite additional pipeline stages, continuous optimization improves almost all benchmarks. Speed ups range from 0.99 to 1.27. On the low side, bzip2 and vortex have the largest baseline IPCs, and both have few branch mispredictions. Such high baseline performance leaves little room for improvement. For the SPECfp benchmark ammp (amp), IPC is low due to data cache misses, i.e., improvements are dwarfed by memory miss cycles. Several benchmarks demonstrate significant improvements. Perl (prl), gap, and eon have the highest percentage of dynamic instructions that can be optimized using the symbolic expression. These benchmarks derive most of their benefit from reducing dataflow height rather than early load address resolution, early branch resolution, or early execution. In contrast, mesa (msa), SPECfp's top performer, derives most of its benefit from memory bypassing.

Table 3 presents characteristics of continuous optimization. *Exec. early* is the percentage of retired instructions executed by the optimizer. *Recov. mispred. brs.* is the percentage of mispredicted branches recovered through optimization. *Ld/st addr. gen.* is the percentage of load and store instructions for which the optimizer generated addresses. *Lds removed* is the percentage of loads converted to move operations by redundant load elimination or store forwarding. *Sts removed* is the percentage of stores removed as silent.

Application Type	Name		Total Insts.	Application Type	Name		Total Insts.
SPECint	bzip2	(bzp)	293M	SPECfp	ammp	(amp)	1000M
	crafty	(cra)	625M		applu	(app)	198M
	eon	(eon)	110M		art	(art)	1000M
	gap	(gap)	500M		equake	(eqk)	1000M
	gcc	(gcc)	284M		mesa	(msa)	1000M
	gzip	(gzp)	869M		mgrid	(mgd)	1000M
	mcf	(mcf)	411M	mediabench	g721 decode	(g721d)	751M
	perlbmk	(prl)	1000M		g721 encode	(g721e)	406M
	twolf	(twf)	595M		untoast (gsm decode)	(untst)	84M
	vortex	(vor)	272M		toast (gsm encode)	(tst)	268M
	vpr	(vpr)	1000M		mpeg2 decode	(mp2d)	164M
					mpeg2 encode	(mp2e)	1000M

Table 1. Experimental workload.

Pipeline	Fetch/Decode/Rename 4 insts/cycle, Retire 6 insts/cycle, 128 physical regs, 160 max inflight insts, 20 (22 for continuous optimization) cycles (min) for BR res (if not executed early)
Branch Predictor	64Kbit hybrid predictor (RAS, global, local, and loop predictors), 4K-entry BTB
Load/Store Queues	48 entry load queue, 48 entry store queue (memory dependence prediction using store sets)
Scheduler	3 16-entry schedulers (int, complex int, fp), 2 8-entry schedulers (load, store) – speculative wakeup and instruction replay
ExeUnits	4 Simple IALUs,1 Complex IALU, 2 FPALUs, 1 Ld Agen, 1 St Agen
L1 I Cache	64KB, 4-way assoc., 64B line size, 1 cycle
L1 D Cache	32KB, 2-way assoc., 32B line size, 1 read port, 1 write port, 2 cycles
L2 Unified Cache	1MB, 2-way assoc., 128B line size, 10 cycles
Memory	100 cycle latency
Optimizer	MBC of 32 entries, 4 rd/4wr ports, 1 cycle value feedback delay

Table 2. Simulated machine configuration.

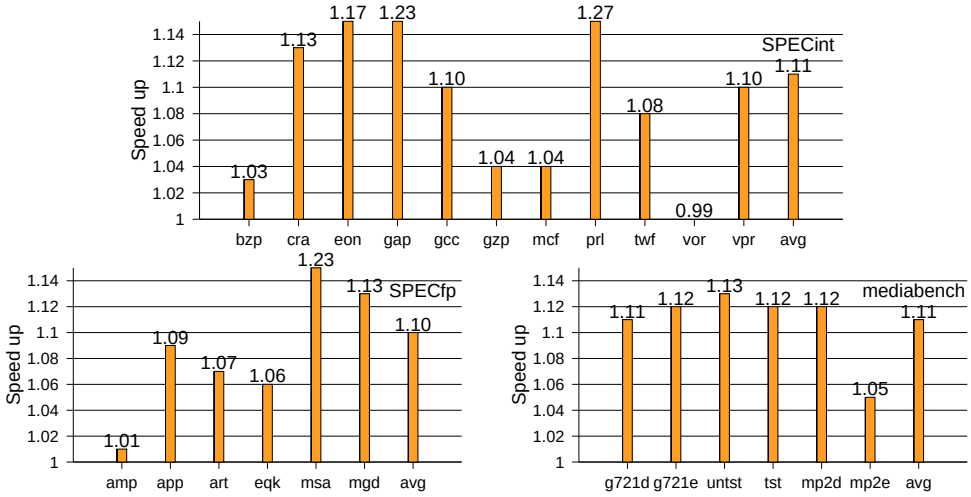

Figure 5. Speedup of continuous optimization over baseline.

Benchmark	exec. early	recov. mispred. brs.	ld/st addr. gen.	lds removed	sts removed
SPECint	32%	16%	61%	13%	2%
SPECfp	31%	45%	76%	23%	2%
mediabench	36%	26%	93%	27%	2%
Average	**33%**	**29%**	**76%**	**21%**	**2%**

Table 3. Performance characteristics of continuous optimizer.

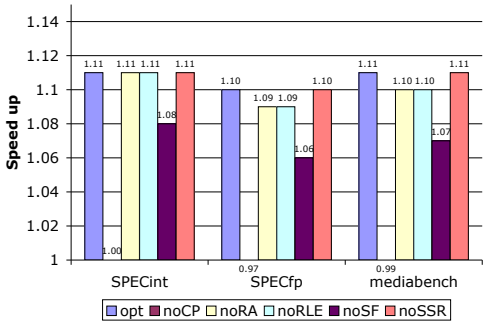

Figure 6. Performance contributions of individual optimizations.

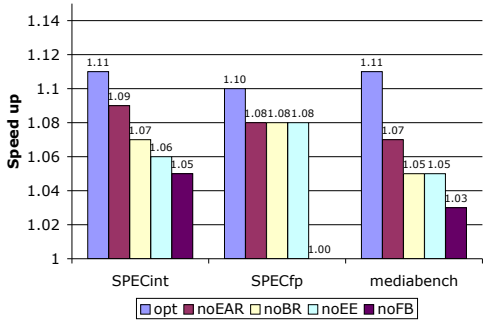

Figure 7. Performance contributions from continuous optimization benefits.

5.2. Performance of safe optimization

By default, continuous optimization removes load and store instructions. Although optimizations are non-speculative, real systems may be unable to remove memory accesses if volatile locations or shared memory are not identifiable. To examine performance with only safe optimizations, we evaluated continuous optimization performance while forcing all memory instructions to access the cache, verify the optimization, and initiate a recovery similar to branch misprediction recovery if a misspeculation occurs. Because we only have single threaded applications and our optimizations are non-speculative with respect to a single thread, misspeculations never occur in our experiments. Even though removing memory access instructions is important, performance improvements are large despite conservative optimization. Performing only safe optimizations reduced average speed up by 0.02x to 0.03x.

5.3. Contributing performance factors

We now look at performance contributing components of continuous optimization. Specifically, we measure contributions of the five optimizations and the impact of continuous optimization's various benefits.

5.3.1. Individual optimization impact. To measure individual optimization impact, we gathered five measurements where only one optimization was turned off for each measurement. Because of the co-dependent nature of optimizations, we found that evaluating performance when individual optimizations are disabled is a meaningful way of identifying the individual optimizations' contributions. Figure 6 shows the results from this experiment. `opt` has all optimizations on, `noCP` has no constant propagation, `noRA` has no reassociation, `noRLE` has no redundant load elimination, `noSF` has no store forwarding, and `noSSR` has no silent store removal. `noCP` indicates that constant propagation is the most important optimization. Without it, memory addresses can not be computed, RLE/SF/SSR do not work, instructions cannot execute early, and branches cannot resolve early. Store forwarding, demonstrated by `noSF`, is the second most important optimization.

5.3.2. Impact of continuous optimization benefits. The benefits of continuous optimization are early load address resolution, early branch resolution, early instruction execution, and dataflow height reduction. Because dataflow height reduction is an integral part of continuous optimization, we begin with our default continuous optimizer and successively remove all other benefits until it is the only remaining benefit. Although feedback is not a benefit, we additionally remove it to demonstrate the importance of value feedback to reducing dataflow height. Figure 7 demonstrates the impact of successively removing benefits. `opt` is the default configuration with all benefits. `noEAR` removes early load address resolution. `noBR` additionally removes early branch resolution. `noEE` additionally removes early execution. `noFB` additionally removes value feedback. For SPECint, dataflow height reduction is roughly half of the improvement. For SPECint and mediabench, nearly all factors contribute to overall performance. For SPECfp, improvements are primarily from memory bypassing, which is significantly improved by value feedback.

6. Performance sensitivities

In this section, we evaluate continuous optimization's sensitivity to various implementation parameters.

6.1. Dependence depth

Since multiple instructions are processed in parallel (e.g., four in a four-wide machine), it may be difficult to optimize instructions when dependences exist within a rename bundle. By default, only the first in a chain of dependent additions is optimized, and, for chained memory accesses, only the first access in a chain queries the MBC. We now measure missed opportunities resulting from our conservatism by evaluating three additional scenarios: (1) one level of chained additions, (2) three levels of chained additions, and (3) three levels of chained additions and one chained memory operation. Figure 8 has

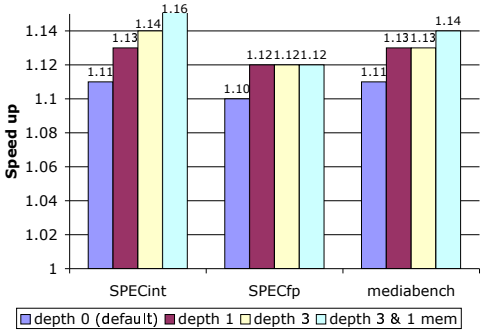

Figure 8. Importance of processing dependent instructions in parallel.

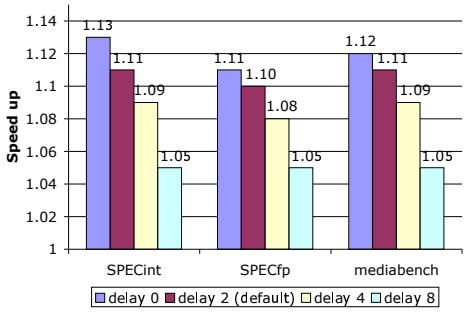

Figure 9. Optimizer latency sensitivity.

bars for the default optimizer and for the three new scenarios. For SPECfp, performance improves only with one level of dependence. SPECint and mediabench, however, benefit from processing multiple dependent instructions in parallel. Note that compiler scheduling of rename bundles could potentially provide comparable performance to the most aggressive case without additional hardware complexity. For ISAs with smaller register sets, like x86, processing multiple levels of dependence will be more important.

6.2. Optimizer latency

Continuous optimization has been assumed to add two stages to a twenty stage pipeline. We now evaluate the sensitivity to this latency. Since optimization occurs in the pipeline, this latency elongates the branch recovery critical loop, potentially reducing performance. As shown in Figure 9, continuous optimization performance varies based on the additional pipeline stages, but even at eight additional pipeline stages (i.e., $\frac{2}{5}$ of the baseline branch recovery cost), there is still an average 1.05 speed up for all benchmark suites. For all experiments including this one, instructions not in the same rename bundle as their producer can be optimized.

6.3. Value feedback latency

Implementation constraints may complicate value feedback, potentially making the transmission take multiple cycles. If delay is too high, values may no longer be useful. By default, we assume that once an instruction executes, its value requires one cycle to be transmitted to the optimizer before it can be used for optimization. We evaluated delays of zero, five, and ten cycles. Only at ten cycles is performance affected, and, even then, the impact is minor (reduces average speed up by 0.01x)[4]. The source of this latency tolerance is because physical registers usually are either referenced for a long period of time or not at all. Recall the example from Section 2.5: the initial loop counter and array address load were in the loop's prologue. Optimization extended their live range to include all iterations. Under such a scenario, the transmission delay of the values has little impact. We also observed how often forwarded values were used and found that only 21% of executed values from the out-of-order part of the pipeline update optimization table expressions, which indicates opportunity for simplifying the value forwarding network or reducing power dissipation by forwarding only necessary values. We leave this study for future work.

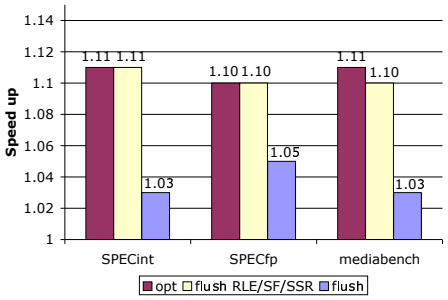

Figure 10. Importance of optimizer state recovery following branch mispredictions.

6.4. Misprediction recovery

We have not yet discussed the impact of branch mispredictions on optimizer state. By default, the CP/RA and RLE/SF/SSR tables are recovered following branch mispredictions. Because CP/RA is an extension of the RAT, adding state recovery for it may be easier than adding state recovery for RLE/SF/SSR. Therefore, in Figure 10 we show both the impact of recovering only the CP/RA table and the impact of not recovering any optimization state. Flushing the CP/RA table on branch mispredictions significantly reduces the effectiveness of continuous optimization. Recovering the RLE/SF/SSR table only impacts average performance for mediabench.

7. Related work

Prior works that significantly overlap with continuous optimization were discussed in Sections 1 and 2.3. Here we discuss only more remotely related works. Physical reg-

4 A five cycle transmission reduced mediabench speed up by 0.01x.

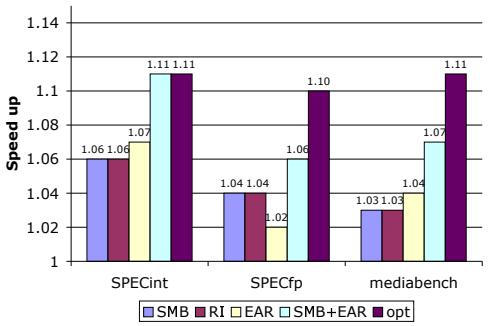

Figure 11. Comparison with competing technologies.

ister reuse [15] identifies instructions producing identical results to prior instructions and maps their outputs to the prior instructions' outputs. A part of continuous optimization implements a form of safe physical register reuse. Flea-Flicker [3] and continuous optimization both pre-execute instructions early in the pipeline, but Flea-Flicker's primary purpose is to allow in-order machines to absorb cache misses. Continuous optimization's primary purpose is to optimize the instruction stream; early execution is a subset of what continuous optimization offers. Physical register inlining [18] incorporates some physical register values into the RAT. Continuous optimization's value feedback is a similar concept. RENO [25] is a follow-on implementation of continuous optimization (in its original form [10]) that provides a subset of the benefits discussed here and adds instruction fusion [11, 14].

There are complimentary works that could provide a synergistic effect when combined with continuous optimization. Scheduling optimizations [16] can be layered onto continuous optimization. In [26], power consumption is reduced by avoiding architectural register file updates for short-lived values. Early register deallocation [19, 22] could be more effective with continuous optimization because of early execution.

Although continuous program optimization [17, 28] has a similar name to our work, it is a different topic. Continuous program optimizers are software systems that tune the performance of applications.

7.1. Quantitative comparison to prior work

Continuous optimization subsumes some previous in-pipeline optimization techniques. We now compare continuous optimization with speculative memory bypassing, reverse integration, and early load address resolution. To ensure a fair comparison, we implemented the prior works optimistically. For example, speculative memory bypassing and reverse integration are allowed to perfectly bypass between memory operations that are still in the pipeline. Reverse integration is also allowed to perfectly bypass opposite operation-pairs that are still in the pipeline. Early load address resolution optimizes all dependence levels, handles all expressions identified by the prior work, and also handles expressions of the form `(reg<<scale)±offset` (the original work only handled `reg±offset` operations). Figure 11 compares these works with our default continuous optimizer, which does not perfectly bypass memory or optimize multiple dependence levels within a pipeline stage. `SMB` is speculative memory bypassing, `RI` is reverse integration, `EAR` is early load address resolution, `SMB+EAR` is speculative memory bypassing combined with early load address resolution, and `opt` is our default continuous optimizer. Although prior works used different simulators (even different ISAs for early load address resolution), we feel that our evaluations of prior work are at least comparable to (if not better than) the results provided by the authors. With the exception of `SMB+EAR` on SPECint, continuous optimization outperforms all others. Continuous optimization's performance advantage can be attributed to more aggressive optimization and early branch resolution.

8. Summary

In this paper, we present and evaluate continuous optimization. Our table-based optimizer integrates with the rename stage of a dynamically-scheduled processor. It performs dataflow optimizations by representing register values symbolically. The optimizer performs constant propagation, reassociation, redundant load elimination, store forwarding, and silent store removal. We enhance optimization with values generated during execution. The optimizer's hardware budget is modest, requiring approximately 2K bytes of additional multi-ported storage and 4 simple ALUs.

Continuous optimization produces speed ups ranging from 0.99 to 1.27 on a pipelined processor similar to the Pentium 4. The optimizer provides many benefits: it reduces dataflow height, executes many instructions, resolves branch mispredictions, and determines memory instruction addresses and values at rename. Each of these components is important to the overall performance improvement of continuous optimization.

9. Acknowledgements

We thank the other members of the Advanced Computing Systems group. This material is based upon work supported by the National Science Foundation under Grant Nos. 0092740 and 9984492 with very gracious support from Intel Corporation and Sun Microsystems.

References

[1] Y. Almog, R. Rosner, N. Schwartz, and A. Schmorak. Specialized dynamic optimizations for high-performance energy-efficient microarchitecture. In *Proceedings of the 2nd International Symposium on Code Generation and Optimization*, pages 137–148, 2004.

[2] V. Bala, E. Duesterwald, and S. Banerjia. Transparent dynamic optimization: The design and implementation of Dynamo. Technical Report HPL-1999-78, Hewlett-Packard Laboratories, June 1999.

[3] R. D. Barnes, E. M. Nystrom, J. W. Sias, S. J. Patel, N. Navarro, and W. mei W. Hwu. Beating in-order stalls with "flea-flicker" two-pass pipelining. In *Proceedings of the 36th Annual International Symposium on Microarchitecture*, pages 387–398, 2003.

[4] M. Bekerman, A. Yoaz, F. Gabbay, S. Jourdan, M. Kalaev, and R. Ronen. Early load address resolution via register tracking. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 306–315, 2000.

[5] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Proceedings of the 1st International Symposium on Code Generation and Optimization*, pages 265–275, 2003.

[6] J. A. Butts and G. S. Sohi. Dynamic dead-instruction dectection and elimination. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 199–210, 2002.

[7] H. Chen, W.-C. Hsu, J. Lu, P.-C. Yew, and D.-Y. Chen. Dynamic trace selection using performance monitoring hardware sampling. In *Proceedings of the 1st International Symposium on Code Generation and Optimization*, pages 79–90, 2003.

[8] W. Chen, S. Lerner, R. Chaiken, and D. Gilles. Mojo: A Dynamic Optimization System. In *3rd Workshop on Feedback-Directed and Dynamic Optimization*, 2000.

[9] B. Fahs, S. Bose, M. Crum, B. Slechta, F. Spadini, T. Tung, S. J. Patel, and S. S. Lumetta. A performance characterization of a hardware mechanism for dynamic optimization. In *Proceedings of the 34th Annual International Symposium on Microarchitecutre*, 2001.

[10] B. Fahs, T. Rafacz, S. J. Patel, and S. S. Lumetta. Continuous optimization. Technical Report UILU-ENG-04-2207, University of Illinois at Urbana-Champaign, August 2004.

[11] D. H. Friendly, S. J. Patel, and Y. N. Patt. Putting the fill unit to work: Dynamic optimizations for trace cache microprocessors. In *Proceedings of the 31st Annual International Symposium on Microarchitecture*, pages 173–181, 1998.

[12] A. Hartstein and T. R. Puzak. Optimum power/performance pipeline depth. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 117–128, 2003.

[13] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The microarchitecture of the Pentium 4 processor. *Intel Technology Journal*, Q(1), 2001.

[14] Q. Jacobson and J. E. Smith. Instruction pre-processing in trace processors. In *Proceedings of the 5th International Symposium on High Performance Computer Architecture*, 1999.

[15] S. Jourdan, R. Ronen, and M. Bekerman. A novel renaming scheme to exploit value temporal locality through physical register reuse and unification. In *Proceedings of the 31st Annual International Symposium on Microarchitecture*, pages 216–225, 1998.

[16] I. Kim and M. H. Lipasti. Implementing optimizations at decode time. In *Proceedings of the 29th annual international symposium on Computer architecture*, pages 221–232, 2002.

[17] T. Kistler and M. Franz. Continuous program optimization. *ACM Transactions on Programming Languages and Systems*, 25(4):500–548, July 2003.

[18] M. H. Lipasti, B. R. Mestan, and E. Gunadi. Physical register inlining. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, 2004.

[19] J. Martinez, J. Renau, M. Huang, M. Prvulovich, and J. Torrellas. Cherry: Checkpointed early resource recycling in out-of-order microprocessors. In *Proceedings of the 35th Annual International Symposium on Microarchitecture*, 2002.

[20] M. C. Merten, A. R. Trick, R. D. Barnes, E. M. Nystrom, C. N. George, J. C. Gyllenhaal, and W. mei W. Hwu. An architectural framework for run-time optimization. *IEEE Transactions on Computers*, 50(6):567–589, June 2001.

[21] A. Moshovos and G. S. Sohi. Speculative memory cloaking and bypassing. In *Proceedings of the 32nd Annual International Symposium on Microarchitecture*, 1999.

[22] M. Moudgill, K. Pingali, and S. Vassiliadis. Register renaming and dynamic speculation: An alternative approach. In *Proceedings of the 26th Annual International Symposium on Microarchitecture*, pages 202–213, 1993.

[23] S. Önder and R. Gupta. Load and store reuse using register file contents. In *Proceedings of the 15th International Conference on Supercomputing*, pages 289–302, 2001.

[24] V. Petric, A. Bracy, and A. Roth. Three extensions to register integration. In *Proceedings of the 35th Annual International Symposium on Microarchitecture*, pages 37–47, 2002.

[25] V. Petric, T. Sha, and A. Roth. RENO: A rename-based instruction optimizer. Technical Report MS-CIS-04-28, University of Pennsylvania, December 2004.

[26] D. Ponomarev, G. Kucuk, O. Ergin, and K. Ghose. Isolating short-lived operands for energy reduction. *IEEE Transactions on Computers*, 53(6):697–709, June 2004.

[27] G. S. Tyson and T. M. Austin. Memory renaming: Fast, early and accurate processing of memory communication. *International Journal of Parallel Programming*, 27(5):357–380, 1999.

[28] R. W. Wisniewski, P. F. Sweeney, K. Sudeep, M. Hauswirth, E. Duesterwald, C. Cascaval, and R. Azimi. Performance and environment monitoring for whole-system characterization and optimization. In *Proceedings of the 1st Watson P=ac^2 Conference*, pages 15–24, October 2004.