Dynamic Parallelization and Vectorization of Binary Executables on Hierarchical Platforms

Efe Yardımcı Michael Franz EYARDIMC@UCI.EDU FRANZ@UCI.EDU

Donald Bren School of Information and Computer Sciences University of California, Irvine Irvine, CA 92697 USA

Abstract

As performance improvements are being increasingly sought via coarse-grained parallelism, established expectations of continued sequential performance increases are not being met. Current trends in computing point toward platforms seeking performance improvements through various degrees of parallelism, with coarse-grained parallelism features becoming commonplace in even entry-level systems.

Yet the broad variety of multiprocessor configurations that will be available that differ in the number of processing elements will make it difficult to statically create a single parallel version of a program that performs well on the whole range of such hardware. As a result, there will soon be a vast number of multiprocessor systems that are significantly under-utilized for lack of software that harnesses their power effectively. This problem is exacerbated by the growing inventory of legacy programs in binary executable form with possibly unreachable source code.

We present a system that improves the performance of optimized sequential binaries through dynamic recompilation. Leveraging observations made at runtime, a thin software layer recompiles executing code compiled for a uniprocessor and generates parallelized and/or vectorized code segments that exploit available parallel resources. Among the techniques employed are control speculation, loop distribution across several threads, and automatic parallelization of recursive routines.

Our solution is entirely software-based and can be ported to existing hardware platforms that have parallel processing capabilities. Our performance results are obtained on real hardware without using simulation.

In preliminary benchmarks on only modestly parallel (2-way) hardware, our system already provides speedups of up to 40% on SpecCPU benchmarks, and near-optimal speedups on more obviously parallelizable benchmarks.

1. Introduction

Sequential execution performance is hitting performance roadblocks on several fronts. First, it is becoming increasingly difficult to further increase clock frequencies as on-chip power densities are rapidly approaching hard physical limits that exhaust the range of available cooling options. Increases in die densities also introduce wire delays as a design constraint.

Furthermore, architectural features such as branch prediction, out-of-order execution and multiple-issue processors that have previously been successful in exploiting ILP tend not to scale very well. As a result, designers have been focusing on coarser levels of parallelism, such as simultaneous multithreading (SMT) and chip multiprocessing (CMP). The first

commercial microprocessors offering such features are already available and the next few years are likely to bring a rapid expansion of multiprocessing capabilities in off-the-shelf microprocessors at multiple levels.

The broad variety of multiprocessor configurations that will be available that differ in the number of processing elements will make it difficult to statically create a single parallel version of a program that performs well on the whole range of such hardware. As a result, there will soon be a vast number of multiprocessor systems that are significantly under-utilized for lack of software that harnesses their power effectively. This problem is exacerbated by the growing inventory of legacy programs in binary executable form with possibly unreachable source code.

Another issue that will increasingly acquire importance is how to optimally map coarsely parallel tasks to processing elements with varying degrees of proximity. A typical scenario will involve an n-way SMT processor on an m-way CMP with possibly multiple chips on a single platform; or even a chip containing heterogeneous processing elements, such as the Cell processor [1]. On such a platform, the optimal mapping of parallelized tasks from an application (these may be successive loop iterations, partitioned loops, or recursive procedure calls) to processing elements may change significantly with program inputs; different regions of an application may also require different mappings. Even a specific code region may be parallelized to different processing elements during different instances in the course of a program's execution.

Our approach to overcoming these problems is to perform the step of mapping a problem to actual parallel resources only at run-time, in a thin software layer running on the target execution platform. This thin software layer is a virtual machine, that executes on top of explicitly parallel processors and performs dynamic recompilation of the code stream at run-time. These parallel processors may be coupled by shared memory or may reside on the same physical processor in the case of an SMT processor.

While a program can be written in a way to facilitate better utilization by our runtime system, our dynamic recompilation strategy is geared towards binary parallelization and vectorization. This involves transformation of an existing binary executable into a structure that makes it possible to be optimized at run-time to make optimal use of existing parallel resources. This approach two major advantages: First, programs can continue to be designed following the familiar sequential model and don't need to be rewritten radically. Second, only one version of the binary is necessary for good execution performance across a wide range of hardware, including uniprocessors. The latter advantage also implies a new lease of performance for older legacy software.

Figure 1) depicts a typical scenario of a system with multiple parallel resources being extended by our software layer. After a once-only static analysis and transformation phase, a binary executable optimized for a uniprocessor system is able to utilize the available parallel resources automatically.

The rest of this paper first discusses related work in Section 2. A description of our implementation, including system inputs, the Intermediate Representation (IR) and our dynamic recompilation methodology is presented in detail in Section 6. Section 7 describes our evaluation methodology and presents our results. Finally, in Section 8 we present our conclusions and describe the future direction of our research.

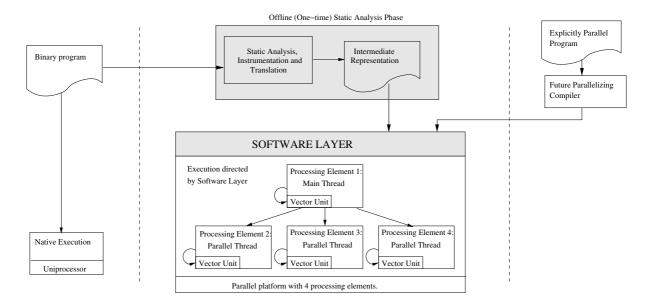


Figure 1: Binary program has been compiled for direct native execution on a uniprocessor (at left). In order to utilize untapped medium-grained parallelism hidden in the program (middle), we run it through a static analyzer. The output of the analyzer is an intermediate structure, which can be parallelized easily at runtime by a lightweight software layer implementing a runtime compilation system. Future compilers for explicitly parallel programming languages (at right) may also target the system directly, by emitting a compatible intermediate structure.

2. Related Work

Our work is based on advances made by two substantial bodies of related work: software-based dynamic compilation and exploitation of mid-level parallelism at a granularity between ILP and cluster computing.

2.1. Software-Based Dynamic Compilation

Executing programs on top of a software layer between the program and the hardware platform has many advantages, including increased capabilities for monitoring and exploiting program behavior.

Dynamo, by Bala et al. [2] by is a well-known system using a software layer between the application and the hardware to improve program performance. Dynamo monitors the program to identify "hot spots" and optimizes these frequently executed regions, which are placed in a software cache. Subsequent invocations of such regions use the optimized copy in the software cache, which over time hold all the frequently executed code. The ADAPT system by Voss et al. [3] is similar, obtaining performance increases with a system that called standard static compilers on remote machines (changing optimization flags and strategies) during program execution.

The DyC system by Grant et al. [4] introduced "selective dynamic compilation" to transform parts of applications identified during profiling stages, using runtime information. Using profiling stages to identify optimization targets, templates are created for each target region that are then used to emit specialized code at runtime that reflect near-constant variable values.

Kistler et al. [5] implemented a system that first monitors the usage of instance variables within dynamically allocated (heap) objects and then carries out runtime object layout changes. The system places all object fields with high access frequencies in the same cache line, improving cache performance. A second, more aggressive optimization orders the fields to reflect the cache refill mechanism.

The Master/Slave Speculative Parallelization paradigm by Zilles and Sohi [6] executes a "distilled" version of the program on a master processor; the results computed by this approximate process are verified by slave processors. At defined places, the master process spawns tasks onto slave processors which execute the original version of the given code. As the distilled version itself has no correctness constraints, performance increases are obtained over the original program.

Another advantage of executing through a software layer is the ability to execute nonnative binaries. This is useful for utilizing a platform customized for a certain feature (such as reduced power consumption) without breaking binary backward compatibility the DAISY just-in-time compiler [7], for example, translates PowerPC binaries into a form acceptable to a VLIW processor [8].

The Crusoe processor by Transmeta [9] is an example for both continuous runtime optimization and binary translation. Crusoe is a relatively simple VLIW processor with a software layer continuously translating and optimizing x86 instruction sequences. While its performance has not surpassed mainstream x86 processors, lower power consumption has been obtained in part due to the software layer performing the duties of complicated conventional hardware.

Intel's IA-32 EL [10] is a software layer that executes x86 binaries on Itanium processors. IA-32 EL first uses a fast, template-based translation approach to generate instrumented native Itanium code. That way it can observe program behavior longer than traditional interpretive approaches. These observations are then used for optimization during runtime "hot code optimization" stages.

Recently, several software-based techniques that perform dynamic speculative execution have been introduced [11, 12, 13]. These require source-code access to the applications and generally rely on profiling stages or manual manipulation of application code for parallelization. Our system takes stripped binary executables as inputs and implements optimizations fully automatically.

2.2. Mid-Level Parallelism

Our solution targets a mid-level parallelism between ILP and cluster computing. In general, one distinguishes among solutions that preserve the illusion of sequential execution (*implicitly parallel*) and those that expose the parallelism towards the programmer/compiler (*explicitly parallel*). The latter designs are generally effective for throughput-oriented workloads, but increase the complexity of the programming model.

Our system is an implicitly parallel design. Unlike the majority of the proposed solutions in this space, we target off-the-shelf processors rather than processors with novel hardware mechanisms (ideal target woulds be the Hydra [14] or Cell [1] processors). Other researchers have come up with some highly interesting new hardware designs to support implicit medium-grained parallelism. None of these designs have actually been built so far—research results have been obtained through simulations.

Multiscalar Processors [15] partition the Control-Flow Graph (CFG) of a program into tasks, which are sequences of contiguous dynamic instructions, and attempt to execute these tasks at different parallel units. Task creation and execution are controlled by a hardware sequencer, and an additional hardware buffer is needed to hold speculated memory operations and checking dependences.

Tsai et al. [16] introduced *superthreading*, a speculative parallelization technique in which hardware mechanisms for dependence-checking and value-forwarding allow the static compiler to be less conservative in designating concurrent threads. While his method requires special hardware support, the compiler also plays a crucial role in creating threads.

Trace processors [17] build on the *trace cache* idea, caching frequently executed sequences of noncontiguous instructions (traces) to obtain more efficient instruction fetching. Relying on hardware-based value prediction for the live-in values at thread entries, trace processors attempt to speculatively execute traces on multiple processing elements.

3. System Overview

The problem of efficiently parallelizing precompiled programs devolves into a three-part problem: converting a binary executable into a form that is easy to monitor and recompile, the runtime identification of optimizable regions, and the runtime recompilation of identified regions.

As displayed in Figure 2, our system's parallelization strategy is geared towards splitting a loop's iteration space and distributing the tasks to available processing elements, instead of

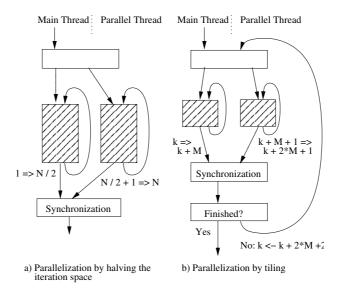


Figure 2: Two parallelization methods. Method (a) is preferred and implemented when the iteration size is known deterministically upon entering the loop. Method (b) uses a form of loop tiling when the loop iteration size is indeterministic.

distributing consecutive iterations across threads. Although this results in lower overheads, it requires being able to know how many iterations a loop is going to execute when it has begun executing. For loop bodies containing break statements, the loop iteration count cannot be precisely determined even at runtime, so parallelization proceeds by "tiling" of the loop into fixed-size chunks, as shown in part (b) of Figure 2 ¹.

Tiling of a loop introduces more synchronization phases than is necessary, where each thread needs to verify that all other threads are finished with their assigned tiles before proceeding to the next tile. However, choosing a tile size (which can be a fixed value for a given platform) large enough to overcome the synchronization costs for the platform enables overall improvements in program performance. For systems with large communication overheads, such as those with processing elements on separate chips, the tile size will be necessarily large, and for systems with lower communication latencies it can be significantly lower.

A critical property of our system is that the only speculation that is carried out is based on control flow; there is no speculation on data. We speculate that once a region is identified as having high loop-iteration counts (and is therefore a feasible candidate for parallelization), it will display similar characteristics in subsequent accesses. We are aware of the fact that programs have temporal "phases" with different execution characteristics; however, previous research [18] (and our observations) show that loop iteration counts display strong value locality during the execution of a program.

As there is no data speculation, the legality-checking phase that occurs after a region is identified as a possible target for recompilation involves data dependence analysis. The

^{1.} It should be noted that recursive procedures pose certain problems in identification and parallelization; these are also dealt with by our system.

main criteria for a loop to be considered parallelizable include: all induction registers having determinate fixed strides, stores to memory and induction registers postdominating the beginning of the loop body, and no potential across-iteration data dependences. While these may seem to constrain the amount of parallelizable regions that can be discovered, we show that the resultant coverage can yield significant performance improvements.

4. Static Analysis of Binary Executables

As our system is completely source-language agnostic, the binary code is converted into a higher-level IR during a static translation stage. With this one-time (per application) translation stage, we are able to address a wide range of programs.

The purpose of the IR is to facilitate fast dynamic recompilation and lightweight profiling of frequently executed code regions. This is accomplished by partitioning the CFG into regions that allow fast code generation. The IR contains various structures to support this aim; these include both global and superblock-level structures.

The global structures include a table that handles execution of indirect branches (explained in detail in Section 4.3.), a superblock-level CFG, and code segment blocks that handle transfer of control flow between the application code and the system code. Superblock-level structures include register def-use information that enables efficient profiling of superblock inputs and a dominator tree for all superblock's basic blocks in order to be able to quickly identify data dependencies. This IR is created only once for a given application and can be employed any number of times with any input.

As our system does not require any special hardware support, it can be deployed on any existing platform that has parallelism capabilities in the form of multiple processing elements or vector units. As the aim of this research is to investigate the benefits of software-directed parallelization and vectorization, we have intentionally left out optimizations such as loop unrolling and code straightening that have been shown to be beneficial.

4.1. Superblock-level Granularity of Optimization

The code-region granularity on which our system implements optimizations is the *superblock* structure, introduced by Mahlke et al. [19]). Their superblock scheduling technique was intended as an across-basic block scheduling mechanism that eliminates some of the complexities associated with trace scheduling [20].

A superblock is a collection of basic blocks where control flow can enter from only one block but may exit from multiple basic blocks, and which does not contain any loops.

The main benefits of having a superblock-level execution granularity is twofold. Firstly, there is the reduction in profiling overhead. Instead of instrumenting each basic block to obtain runtime profile, only the basic block that dominates a group of basic blocks (not going beyond loop boundaries) is instrumented—while still obtaining useful profile information.

An alternative would have been to designate loop bodies as the optimization granularity. However, this would be problematic as different loops may share code segments, as can be seen in Figure 3. It is not clear at load-time what the prevalent loop will be for this piece of code—whether it will take the form of A-B or C-B—, dynamic information is required to identify and optimize a region. Static code duplication would result in excessive code growth, especially if neither loop is frequently executed.

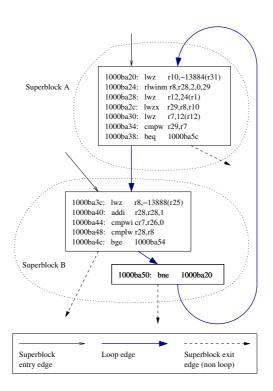


Figure 3: Loop example.

Thus the second benefit of having superblock-level execution is to be able to analyze complex loop bodies without excessive code growth.

Once a superblock —or perhaps a region consisting of a sequence of superblocks— is identified as a region that should be optimized, the region is recompiled to optimize for the displayed dynamic characteristics. A given loop may be parallelized or vectorized (possibly both) depending on various properties such as iteration count, stride predictability, the memory access pattern and platform constraints.

It should also be noted that although our system does not carry out other kinds of runtime optimizations such as strength reduction, code straightening and loop unrolling, these optimizations would also benefit from the superblock-based granularity of our system.

4.2. Control Flow Analysis

Having binary executables as system inputs poses a problem in the extraction of correct superblock structures. The cause of the problem is the presence of indirect branches; by definition, control should enter a superblock at a single location. If superblocks are identified by a naive single-pass method, it is possible to have indirect branches jump to locations that were not recognized as being superblock entries (side-entries).

Although a perfect control flow graph information would be highly desired, perfect coverage is not a necessity—non-superblock-entry indirect branches can be handled and do not cause noticeable performance degradation if they occur rarely. However, obtaining good superblock structure information is important in carrying out effective register def-use and liveness analyses; this too is complicated by the presence of indirect branches.

The usual method for obtaining the CFG from a binary executable is to create an imperfect graph using the branches with explicit targets. In a second step, reaching definition analyses and constant propagation are applied to iteratively refine the graph. Once new indirect branch targets are obtained through reaching-definition analysis, the new edges are inserted into the CFG and a new iteration stage is started. These steps can be repeated until no new edges can be found. However, we see related projects only apply several passes, which is sufficient for their utilization of the CFG (binary instrumentation or decompilation) [21, 22, 23, 24].

To make our runtime monitoring as lightweight as possible, we wanted to obtain as accurate a CFG as possible, to have better information on the availability and liveness of registers. Our initial intent was therefore to repeat the analysis stages until a fixed point would be reached.

However, this technique has very high execution time and memory requirements, especially with static binaries. A naive (albeit completely legal) solution would be to conservatively identify all basic blocks as targets of indirect branches and use the resulting graph as the control flow graph². While this solution would eliminate unseen paths, it would also limit aggressive optimizations. This is an optimization problem: we want to have superblocks as large as possible without causing excessive amounts of side-entries.

We have implemented a heuristic that gives us useful indirect branch target information with reasonable effort, without introducing many spurious control flow graph edges. Our method involves starting with an imperfect CFG, performing iterative reaching definition analysis to create def-use chains and obtaining values of target registers³ at indirect branch locations.

Thus, we get all definitions made by branches that set the target registers as well as those made by explicit load instructions. Instead of restarting the iterations after adding the newly-found edges (which would have been very costly in terms of time and space), we start a new, special iteration phase based on a heuristic which we call the "LR-Heuristic" [25].

This phase results in near-100% dynamic coverage of indirect branches for most Spec-CPU 2000 benchmarks, resulting in zero or negligible number of side-entries to superblocks while minimizing the number of redundant edges added.

The classical reaching definition analysis iteratively calculates:

$$REACHout(b) = GEN(b) \cup (REACHin(b) - KILL(b))$$

 $REACHin(b) = \cup REACHout(p), \ p \in predecessor(b)$

After the completion of this stage, we start the LR-Heuristic phase, where move-from-link-register (mflr) instructions generate and branch-to-link-register (blrx) instructions kill definitions of LR. Then, for each indirect branch reached by a move-to-link-register (mtlr) instruction during the initial reaching definition stage, we find the reaching "LR-Heuristic" definitions generated by move-from-link-register (mflr) instructions. This method is explained in detail in [25].

^{2.} Even though it is still possible after static analysis to have indirect branches jump to locations inside basic blocks, these situations are very rare. Basic block boundaries created through static analysis are almost always identical to those observed at runtime, and our side entry handler mechanism maintains legality of execution for the transgressions that do occur.

^{3.} Indirect branches in the PowerPC architecture are implemented with instructions that jump to the address in one of two special purpose-registers, the LR *Link-Register* and the CTR *Count-Register*.

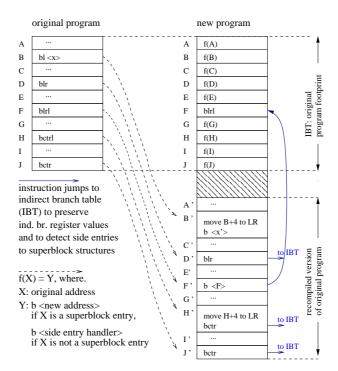


Figure 4: Indirect Branch Table.

As previously stated, perfect coverage is neither a necessity nor an attainable target. We would like to have as correct control flow graph information as possible, but also allow for the possibility of rare side-entries to structures designated as superblocks. We handle those cases with what we call an *Indirect Branch Table (IBT)*.

4.3. Indirect Branch Table

While other types of branches may be statically updated to reflect a new target location, indirect branches cannot be updated this way. Although return instructions such as branch-to-target-register instructions from procedures called with branch-and-set-target-register instructions are safe, these can be broken by pointer arithmetic operations. Also, target register values loaded from memory can be hard to detect. We overcome this problem by maintaining the same values in the indirect branch target registers as in the original program. This is achieved with the use of an *Indirect Branch Table (IBT)* that is mapped to the address space of the original executable, as depicted in Figure 4.

All indirect branches jump *not* to the modified location of the intended target, but to the original target address. This address, part of the IBT, holds an unconditional jump to the address where the modified code now resides. The original LR values are preserved through the insertion of instructions that manually load new target-register values before branch-and-set-target-register instructions, which are then transformed to simple branches.

We use the insight that all indirect branch targets should be superblock entries, and that we know at static-analysis time which addresses are entry addresses. Therefore, we limit normal IBT usage to indirect branches. Except for the results of an overhead-reducing

method (explained in Section 5), at load-time all possible valid IBT accesses are superblock entry accesses. This reduces the problem of detecting side entries to simply having all non-superblock-entry addresses in the IBT contain a jump to a side-entry handler. Any need for trapping superblock exits and querying target addresses is eliminated.

The side-entry handler, in turn, overwrites the original superblock addresses that reside in the memory as part of the IBT with the original instructions. As these events occur highly infrequently, overwriting the addresses and invalidating the corresponding I-cache lines do not cause noticeable performance degradation.

This mechanism allows us to select and optimize regions of a binary program as Single-Entry-Multiple-Exit regions even if the constraints of such regions are occasionally violated (violations, if they occur, are handled in constant time as described above), without incurring noticeable performance penalties.

This is a significant difference from mechanisms such as ATOM or DynInst [26, 27], which also instrument binary executables for more general purposes. Broadly, the IBT allows partitioning of a binary executable into any type of code structure, and ensures the correctness of the chosen execution model by catching and handling violations as they occur.

4.3.1. Memory Utilization

The vast majority of memory utilization is due to the IBT, the system code, and the static and dynamic data segments of the target application. There are other runtime structures as well, required for the implementation of various application-to-runtime system control transfers (handling side entries, profiling, insertion of recompiled code).

The size of the IBT is exactly the size of the code segment of the statically compiled binary executable, as any instruction in the binary executable can be an indirect branch target. The size of the side entry handler (jump table, handler code, etc.) is directly proportional to the number of instructions inside superblocks statically marked as candidates for recompilation. Considering that each instruction requires a 3-instruction jump table entry to reach the irregular entry handler code, the total size of the jump table be can measured in the tens of kilobytes for all cases.

Other runtime structures do not form a significant portion of memory footprint. The IBT and the runtime system code together increase the load-time memory utilization by a factor of at least four, dependent on the size of the binary executable - the size of the IBT and the recompiled code region is proportional to input executable's size but the system code size is fixed.

5. Identification and Classification of Loops

The mechanism for runtime identification of parallelizable regions involves superblocks being linked to the system, or vm-linked, by having the first two instructions overwritten by jumps to a transfer procedure (which also maintains correct program execution).

Regions are detected by employing and monitoring a a FIFO buffer and software cache of superblocks. Once a regular region is detected (for example, a loop involving superblocks A, B and C with body A-B-C-B) the vm link in the first superblock is replaced by a link to a version of the loop that only counts loop accesses (as opposed to transferring control

Benchmark	Total Number	Statically Identified	Dynamic Entries
	of Superblocks	as Feasible	to Feasible Superblocks
gzip	10900	283	19.7%
vpr	12679	342	0.1%
mcf	9415	244	11.7%
parser	13655	300	5.8%
bzip2	10839	286	34.3%
twolf	13388	437	31.0%
perlbmk	14736	478	59.2%
art	12391	322	58.7%
swim	10012	383	32.2%
treeadd	9821	201	97.8%
em3d	9318	188	91.5%

Figure 5: A tiny fraction of an input program's superblocks are statically determined feasible for optimization. Only these few superblocks are ever profiled and dynamically optimized under the control of the Azure system. Yet in many of the benchmarks, a significant proportion of execution time is spent in precisely these optimizable code regions, creating the possibility for performance gains through dynamic optimization.

to the system to maintain the identification structures). The other superblocks are set to native state. If the region is found to be worthy of optimization, recompilation takes place.

An important point to recognize is that not all superblocks need to be instrumented. A great majority of loops that our system rejects later on at runtime can be identified and eliminated during the static analysis stage. This method, which we refer to as post-elimination loop identification drastically reduces the number of superblocks that are profiled, yet the execution time percentage of loops selected with this method stays almost exactly the same. As a result of this static elimination step, only a tiny fraction of an input program's code is actually profiled and dynamically recompiled at run-time. Yet many programs spend a significant proportion of their run-time in precisely the superblocks that were statically identified as being feasible.

With this method, the original code space is not completely overwritten with the IBT; only those superblocks that are not eliminated are overwritten with IBT instructions. Figure 5 shows the number of superblocks that are profiled out of the total number, as well as the access percentages of profiled superblocks.

Figure 6 presents the effect on system overheads that is brought about by this statically exclusion. An interesting result of our approach is that programs with low dynamic coverage of feasible superblocks also have a low instrumentation overhead. For example, consider the vpr benchmark. The complicated control flow of the "hot" code of this benchmark results in low actual execution coverage with our identification mechanism, but as a corollary, our approach also does not do much harm: almost all of the code that is executed eventually at run-time comes from the original program binary, keeping performance very close to what

Benchmark	Overhead of Considering	Overhead of Considering
	All Superblocks	Only Feasible Superblocks
	at Run-Time	at Run-Time
gzip	5.5%	1.9%
vpr	8.1%	1.0%
mcf	6.5%	0.1%
parser	13.8%	0.3%
bzip2	7.8%	1.0%
twolf	7.7%	1.3%
perlbmk	9.4%	1.4%
art	8.9%	1.1%
swim	8.4%	1.2%
treeadd	3.1%	0.4%
em3d	7.1%	0.3%

Figure 6: Static elimination of non-feasible program regions from further consideration significantly reduces dynamic system overheads.

it would have been if the dynamic optimization system had not been present in the first place.

5.1. Special Case: Recursive Loops

In order to be able to exploit parallelism found in recursive loop iterations, we create a special type of superblock just for recursive procedures. Whenever we can see that certain backward edges are dominated by the superblock containing the target of the edges and those edges are branch-with-link instructions, we recognize that the code region forms a recursive procedure and merge the superblocks into a single, larger "recursive superblock".

Except for different recompilation templates and slightly stricter legality checking, recursive superblocks are handled mostly the same way as other superblocks.

5.2. Platform Considerations

Our system uses a heuristic that "accepts" recognized loops as candidates for optimization based on the observed average iteration counts for the first 10 times the loop is entered from elsewhere. If the average iteration count is over a certain threshold, the loop is admitted to the next stage, where further safety checks are made before recompilation occurs. This is a tunable parameter; on platforms where there is close proximity between processing elements this threshold is naturally lower, and on platforms having processing elements of different proximities there are multiple thresholds.

5.2.1. Mapping of Tasks to Processing Elements

We have seen through experimentation that an iteration count of 2K is a feasible threshold for a platform having multiple processors connected through main memory. For a platform

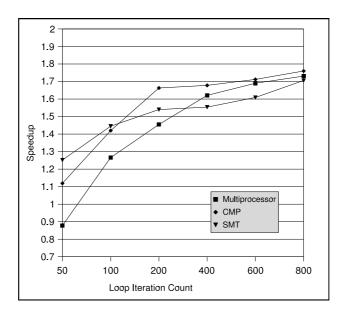


Figure 7: Em3d speedups.

with both SMT (shared physical processor) and CMP (shared L2-cache), we have used thresholds of 100 and 500, respectively. A loop that falls below the CMP threshold but above the SMT threshold would be split and the parallel task sent to a logical processor on the same core to maximize communication speed; a loop with an iteration count above the CMP-threshold would be split to tasks which would reside on different cores on the same chip (and communicate through the L2-cache).

A third threshold would be added if a platform has multiple *chips*, with each chip having multiple cores and each core having logical processors. This would be useful for loops that are so large and have so little communication requirements that it is more efficient to map parallel tasks to different chips, where each task can fully use the available cache.

Figures 7 and 8 illustrate this point. Using the treeadd and em3d benchmarks, which have command-line adjustable loop sizes, we see that different loop size ranges are more suitable for mapping to certain types of processing elements than others. The multiprocessor platform is a dual-processor PowerPC G4, and the SMT/CMP platform is a dual-core POWER5 with each core having 2 SMT processors. Even though it would be more correct to have the same type of processor on all processing elements, we believe the results would be similar.

Two important points should be noted. In treeadd, beyond a certain loop size, it is better to map a parallel task to a multiprocessor than to different cores on the same chip. This could not be verified on em3d the loop size could not be increased after a certain point. A second important point is that as loop sizes diminish, communication overheads dominate execution time, and it can be more advantageous to map parallel tasks to logical processors on the same chip, regardless of the resultant sharing of functional units. At the smallest loop size, parallelizing em3d on a multiprocessor platform actually decreases performance, whereas improvements can still be obtained at an SMT-level mapping (and to a lesser degree on a CMP-level mapping).

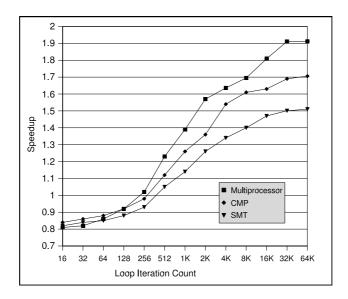


Figure 8: Treeadd speedups.

These thresholds only apply to the dynamic parallelization of loops. A loop with an iteration count less than any threshold can instead be vectorized, depending on the existence of vector extensions on the processing elements. If the loop is not selected for either optimization (possibly because of a low iteration count and irregular strides in memory accesses) it is "released" to the application and never profiled again.

6. Dynamic Recompilation

6.1. Recompilation for Parallelization

The mechanisms that have been described so far lead to critical junctions during the course of an application's execution where the system decides that it has obtained enough information about a given loop region to perform an action on it.

A loop is released immediately if a (highly conservative) legality checking phase cannot verify that the transformation necessary for parallelization is safe..

The most critical property during this decision mechanism is the iteration count, which has already been discussed in this paper. For certain loops this information can be obtained very fast, as the IR obtained during the Static Analysis stage holds stride and definition information for use'd registers. This is the case for loops that have a fixed stride, comparison with a constant value or register, and no internal breaks. Loops that have break instructions need to be profiled with a more heavy-handed approach involving counting each iteration until the loop is terminated, either by a break instruction or by reaching the end of the iteration space.

Part A of Figure 9 displays the additional instructions inserted in a parallelized loop, mostly to calculate and send the register values of the parallel threads.

Also displayed in Part B of Figure 9 are the differences in parallelizing recursive loops. Firstly, the parallel thread is not activated at the entry of the superblock but is called when

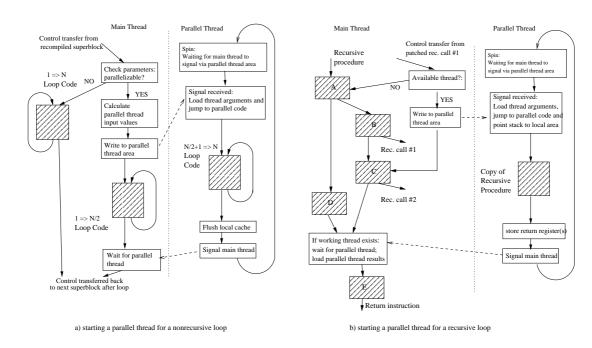


Figure 9: Starting parallel threads.

the first recursive call is made once inside the superblock. This call is intercepted by the runtime system which triggers the passing of a task vector (code address, arguments, return registers) to a free thread, if any exists at that point. The arguments are not modified as they would be in a loop-splitting transformation, but the stack pointer is pointed to a local data area to prevent writing into other threads' stacks⁴.

When this is completed the system returns control to the superblock in the main thread, which proceeds inside the superblock body until a second recursive call is made and returned from. The second main difference is the location of the waiting code, which is immediately after the second recursive call. Any return value that was stored by the parallel thread is fetched and stored into the correct register.

6.2. Verifying Legality of Loop Transformations

As our system speculates only on control flow and has no mechanisms for data speculation, loops are only parallelized if they pass a highly conservative legality checking phase.

For example, it is possible to have control-flow structures *inside* a superblock that may prohibit safe parallelization. An example is a memory store inside an if-then block. Currently we release an identified loop back to the application if all stores in the loop's superblocks do not postdominate the superblock entries. This constraint allows us to carry out data dependence tests very quickly (albeit conservatively).

^{4.} It is not hard to learn which register is the stack register, even if we do not know anything about the original compiler of the binary. Most executable file formats have fields noting the Application Binary Interface (ABI); and initialization codes at the entry point of an application can be analyzed to obtain the stack register.

Another constraint is having constant, predictable strides for superblock input registers. This is important not only for identifying memory accesses without loop-carried dependences but also to be able to split a loop's iteration space: given a loop with \mathbb{N} iterations, we need to be able to determine loop inputs $\mathbb{N}/2 + 1$ iterations ahead for a parallel task. This is not possible without data speculation for loops where index variable are loaded from memory at each iteration.

Theoretically, these constraints could be significantly loosened by taking advantage of value and stride prediction methods. However, our initial explorations concerning profiling for stride locality have not produced immediately usable results. We plan on addressing this topic more aggressively in the future.

Across-iteration dependences are also analyzed conservatively. Currently, a region is released if we cannot determine that a memory access is made to a location local to the current iteration. While this would seem to prevent optimization of most loops, enough are left for the system to identify and break even on most application. On several applications, significant improvements are obtained.

Recursive procedures have further constraints applied to them before they are accepted for parallelization. These constraints are currently very conservative, and result in the acceptance of basic—yet useful—recursive codes. We mainly check that all stores are to the stack, and are thus independent from stores made by concurrent threads executing different branches of the recursive code.

6.3. Managing Parallel Threads

The nature of the parallel threads is another critical design point. Threads are started at load-time on each extra processing element belonging to the platform. For example, in a 4-way CMP we would have 3 extra parallel threads in addition to the main thread. The current state of our system supports at most 2-way parallelization.

Each thread is tied by our system to a single processing element, a capability supported by most modern operating systems (Linux kernels 2.6 and above provide this facility through the new sched_setaffinity system call). The parallel threads are assigned jobs by the main thread, and spin while polling an argument data area where the main thread writes task vectors (code address, number of arguments, and the arguments).

Following the completion of a task, a parallel thread clears the area written by the main thread to signal its completion and returns to the spinning state. If the loop has no determinate iteration count, is not a recursive loop and is being parallelized in small tiles, the parallel thread increments its arguments itself and continues until an end condition is reached in one thread (this is depicted in part (b) of Figure 2).

6.4. Recompilation for Vectorization

If a loop being analyzed is seen to have an iteration count insufficient for parallelization, it may still be considered to be a candidate for vectorization. If the loop is found to be legally vectorizable after a checking phase, it is recompiled to exploit the vector units of the platform. This checking phase must be done at runtime, as we do not know statically if a loop will be parallelized or vectorized; a non-vectorizable loop may indeed turn out to be

fit for parallelization. Conversely, a loop that may not be parallelizable because of break instructions inside the loop body may be vectorized.

Our system is again very conservative while checking legality for vectorization. The legality criteria are based on whether a loop body fits into one of several legal loop code patterns (such as load-execute-store, load-compare or shift array contents). Once a pattern is matched, recompilation is easy as our legal code patterns have corresponding vectorization templates that are used to emit code (this is very similar to the selective dynamic compilation method of Eggers et al. [4]). As we improve our system, we will keep incorporating more accepted cases into the legality checking phase.

Recompiling for vectorization is simpler than recompiling for parallelization. Loops do not have to be split or tiled, and no interaction with parallel threads is necessary. In fact, vectorization is feasible on any uniprocessor that has a vector extension unit such as AltiVec.

A key point to take care of is the memory alignment. AltiVec vector memory instructions operate on 16-byte boundaries; since loops may start at random alignments it is necessary to add a prologue that loops until a valid 16-byte alignment is reached. Similarly, it is important to limit a vectorized loop so that memory accesses do not reach beyond the range of the original loop. An epilogue code is inserted to execute remaining parts not executed because of this limitation.

7. Evaluation and Results

We have implemented our system on a dual 866MHz G4 processor PowerMac, running a Linux OS with a 2.6.8 kernel, and later ported it to a 1.67 GHz POWER5 system with SMT and CMP capabilities. This enabled us to obtain a picture of how the system will perform on a wide range of parallel hardware, with different proximities between processing elements. The benchmarks were chosen from a mix of SpecCPU 2000 integer (gzip to parser), SpecCPU 2000 floating-point (swim and art) and Olden benchmarks (treeadd and em3d). All benchmarks were compiled with GCC 3.2.2 with full optimization and executed with the benchmarks' reference inputs and command line arguments.

Figure 10 shows the results obtained on the dual PowerPC processor. The striped bars shows the performance results with all the overheads (including recompilation) included, without the recompilations being committed. The most significant thing to note is the almost unnoticeable amount of slowdown. This can be accounted to the post-elimination method of loop identification, along with the superblock-level granularity of instrumentation. The solid bars show the actual performance results over optimized binaries running natively on the platform. In almost all cases we are able to break even (except for vpr, where no loop region was found worthy of optimization).

The Olden benchmarks, having simpler structures with parallelizable loops accounting for most of execution time (as can be seen in Table 1) display the best results. The floating point benchmarks, with regular code operations on matrices or vectors, also show impressive results, achieving speedups of 33% and 20% for art and swim, respectively. As expected, it is harder to find legally and feasibly parallelizable regions in integer applications, though we also average around 5% speedup.

The results also show how recompilation for vectorization can prove useful. Within the integer benchmarks, most of the improvement results were obtained through vectorization.

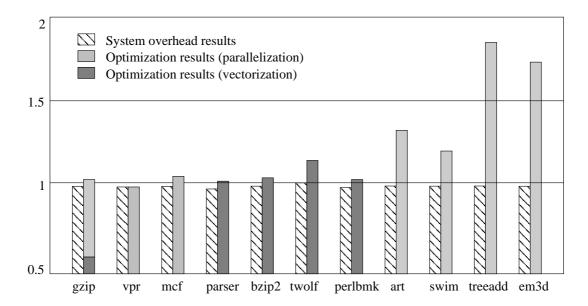


Figure 10: Effects of our system on the performance of statically optimized binary programs, on a dual-processor PowerPC platform. All optimizations break even except for vpr, for which no optimizable region was found. The overall result for vpr corresponds to the overhead of the optimizer.

It is certainly possible to have both vectorization and parallelization applied on the same benchmark, although this only happened on gzip. This is mainly due to the small number of loops selected for optimization, on average less then a dozen loops were parallelized in each benchmark (none in vpr).

The lack of a vector coprocessor similar to AltiVec in the POWER5 processor has considerable impacts on the obtained results. Except for twolf, the integer benchmarks whose performances were able to be improved on the PowerPC platform register a slight slow-down (around 1%) when executed on the POWER5 platform. We can still vectorize twolf to a degree by converting byte-oriented loops to use word-length memory and arithmetic instructions. An interesting result from executing twolf on the POWER5 platform is that even after accounting for the introduced overheads, executing the binary without committing the optimizations results in a *speedup* over an optimized binary running natively. This anomaly is most likely due to reduced I-cache conflict misses resulting from code motion affected by the instrumentation.

In general, mapping parallel tasks to a logical processor on the same core provides smaller returns than when the same tasks are mapped to a different core. This is understandable, as the latter case uses twice the amount of physical hardware to execute the same code. Only in em3d does an SMT-mapping provide better results, when the smaller loop bodies require faster communication.

The results from the multiprocessor platform are generally better then on the SMT/CMP POWER5 platform, but this is mostly due to the presence of the AltiVec coprocessor. Only in treeadd does parallelization on the multiprocessor significantly improve upon the CMP-

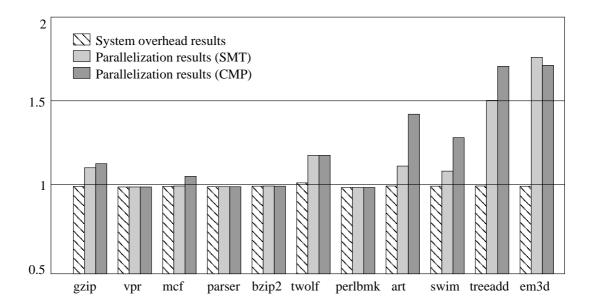


Figure 11: Effects of our system on the performance of statically optimized binary programs, on a dual-processor PowerPC platform. All optimizations break even except for vpr, for which no optimizable region was found. The overall result for vpr corresponds to the overhead of the optimizer.

mapping on the POWER5; this can be explained by the large loop size (16M iterations) due to the reference arguments for the treeadd benchmark.

7.1. Scalability Studies

In order to gauge the performance of the system as the number of processing elements is scaled beyond 2, we have modified the parallelization mechanism to support 4 processing elements.

In general, two factors inhibit scalability. One factor is the effective increase of the parallelization threshold. With the work done per task being halved going from a 2-element to a 4-element system, the iteration count required to surpass the synchronization overhead is doubled, even with constant overheads. Furthermore, on this particular system the communication overhead between parallel threads is also increased⁵.

The performance result displayed in Figures 12 and 13 are in line with these considerations. The results obtained in a 4-way mapping of parallel tasks (4 tasks split into 2 logical processors in 2 cores, hence referred to as 2x2 SMT/CMP mapping) on a POWER5 system are compared with the 2-way mappings to logical and physical processing elements (2-way SMT and 2-way CMP, respectively).

We can see in figure 12 that a speedup plateau is reached when the iteration counts are beyond a certain limit. Speedup results decrease as the iteration averages are reduced,

^{5.} As trends indicate a greater number of processing elements on the same die with on-chip memory controllers, communication overheads may in fact be less intrusive in the future.

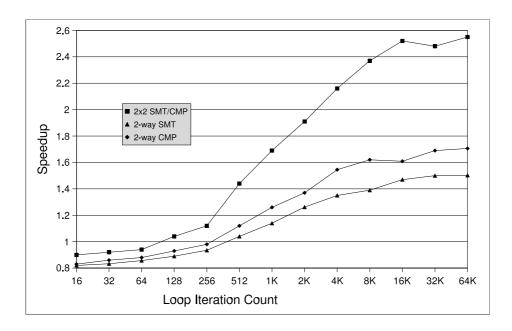


Figure 12: em3d speedups for various loops sizes on 2-way CMP, 2-way SMT and 4-way (2x2) SMT/CMP platforms.

falling below 2 when reduced to 2K (on a 4-way system). With averages of less than 512 iterations, parallelizing treeadd on 4 threads achieves results not significantly better than 2-way parallelization. Similar results have been obtained for em3d.

Certain applications with obviously parallel code will not require run-time monitoring, and certain applications with complex inter-thread communication requirements will still need to be programmed explicitly. What we are claiming is an increased utilization of a portion of available resources to enhance the performances of applications written and compiled without explicit consideration of multiprocessors.

8. Conclusions and Future Work

We have presented a system that improves the performance of sequential, optimized program binaries on explicitly parallel platforms through dynamic parallelization and vectorization. Our system is entirely software-based and targets rapidly emerging off-the-shelf processors with parallel resources.

The task of our software layer is to map, at runtime, desirable portions of a "sequential" program to available parallel hardware to make it run faster. We propose partitioning a binary executable into Single-Entry-Multiple-Exit regions, or superblocks, which simplify dynamic profiling (reducing overheads) and runtime recompilation. The software layer uses control speculation at the granularity of superblocks to extract parallelism from the sequential program.

Our optimization breaks even or achieves significant improvements on a broad range of benchmarks, achieving up to 42% improvement on SpecCPU 2000 floating point, over 85%

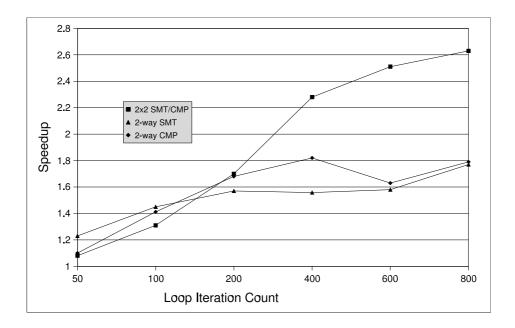


Figure 13: treeadd speedups for various loops sizes on 2-way CMP, 2-way SMT and 4-way (2x2) SMT/CMP platforms.

speedup on Olden and on average a 10% speedup on SpecCPU 2000 integer benchmarks. Perhaps just as importantly, our worst-case results when no optimizable loop has been found (and when all runtime monitoring and identification processes have accrued overhead) is negligible.

Key to the remarkably low overhead is the Single-Entry-Multiple-Exit model of execution regions. The underlying analysis is performed ahead of time in an off-line step. We are able to detect cases that break the region-based execution model with a low constant overhead and correct them on the fly. The analysis permits to determine ahead of time which few parts of an input program can possibly benefit from dynamic profiling and optimization, and thereby makes it possible to exclude the majority of the code from further consideration at run-time.

Based on its low overhead run-time system, worst-case results are only insignificantly worse than if the Azure system was not present at all, in spite of the extra effort spent on run-time monitoring and recompilation.

Continuing onwards from the encouraging results presented here, we aim to continue this research by investigating the effects of scaling on the system. We are especially optimistic on what can be achieved with floating point and media applications.

Other future work include incorporation of data speculation into our decision structure and incorporating our system to specialized hardware solutions such as Hydra; we hope to be able to parallelize a much greater percentage of loops than we can now.

References

- [1] P. Hofstee and M. Day, "Hardware and software architectures for the cell processor," in CODES+ISSS '05: Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis, (New York, NY, USA), pp. 1–1, ACM Press, 2005.
- [2] V. Bala, E. Duesterwald, and S. Banerjia, "Dynamo: A transparent dynamic optimization system," in *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, pp. 1—12, 2000.
- [3] M. J. Voss and R. Eigemann, "High-level adaptive program optimization with ADAPT," *ACM SIGPLAN Notices*, vol. 36, pp. 93–102, July 2001.
- [4] B. Grant, M. Philipose, M. Mock, C. Chambers, and S. J. Eggers, "An evaluation of staged run-time optimizations in dyc," in *Proceedings of the ACM SIGPLAN 1999* Conference on Programming Language Design and Implementation, pp. 293–304, ACM Press, 1999.
- [5] T. Kistler, *Continuous Program Optimization*. PhD thesis, Department of Information and Computer Science, University of California, Irvine, Nov. 1999.
- [6] C. Zilles and G. Sohi, "A Programmable Co-processor for Profiling," in *Proc.* 7th International Symposium on High Performance Computer Architecture, Jan 2001.
- [7] K. Ebcioğlu and E. R. Altman, "DAISY: Dynamic compilation for 100% architectural compatibility," in 24th Annual International Symposium on Computer Architecture, pp. 26–37, June 1997.
- [8] K. Ebcioğlu, J. Fritts, S. Kosonocky, M. Gschwind, E. Altman, K. Kailas, and T. Bright, "An eight-issue tree VLIW processor for dynamic binary translation," in *Proceedings of the 1998 IEEE International Conference on Computer Design*, 1998.
- [9] A. Klaiber, "The technology behind $Crusoe^{TM}$ processors." Transmeta Technical Brief, Jan. 2000.
- [10] L. Baraz, T. Devor, O. Etzion, S. Goldenberg, A. Skaletsky, Y. Wang, and Y. Zemach, "IA-32 Execution Layer: a two-phase dynamic translator designed to support IA-32 applications on Itanium-based systems," in *Proceedings of the 36th International* Symposium on Microarchitecture, pp. 191–201, IEEE, Dec 2003.
- [11] M. Cintra and D. R. Llanos, "Toward efficient and robust software speculative parallelization on multiprocessors," in *Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pp. 13–24, ACM Press, 2003.
- [12] L. Rauchwerger and D. Padua, "The lrpd test: speculative run-time parallelization of loops with privatization and reduction parallelization," in *Proceedings of the ACM SIG-PLAN 1995 conference on Programming language design and implementation*, pp. 218–232, ACM Press, 1995.

- [13] M. K. Prabhu and K. Olukotun, "Using thread-level speculation to simplify manual parallelization," in *PPoPP '03: Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, (New York, NY, USA), pp. 1–12, ACM Press, 2003.
- [14] L. Hammond and K. Olukotun, "Considerations in the design of hydra: A multiprocessor-on-a-chip microarchitecture," Tech. Rep. CSL-TR-98-749, 1998.
- [15] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar, "Multiscalar processors," in 25 Years of ISCA: Retrospectives and Reprints, pp. 521–532, 1998.
- [16] J.-Y. Tsai, J. Huang, C. Amlo, D. J. Lilja, and P.-C. Yew, "The superthreaded processor architecture," *IEEE Transactions on Computers*, vol. 48, no. 9, pp. 881–902, 1999.
- [17] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. Smith, "Trace processors," in *International Symposium on Microarchitecture*, pp. 138–148, 1997.
- [18] M. de Alba and D. Kaeli, "Runtime predictability of loops," 2001.
- [19] W. Hwu, S. Mahlke, W. Chen, P. Chang, N. Warter, R. Bringmann, R. Ouellette, R. Hank, T. Kiyohara, G. Haab, J. Holm, and D. Lavery, "The superblock: an effective technique for vliw and superscalar compilation.," *The Journal of Supercomputing*, vol. 7, no. 1-2, pp. 229–248, 1993.
- [20] J. A. Fisher, "Trace scheduling: A technique for global microcode compaction," *IEEE Transactions on Computers*, vol. 30, pp. 478–490, July 1981.
- [21] J. R. Larus and T. Ball, "Rewriting executable files to measure program behavior," Softw. Pract. Exper., vol. 24, no. 2, pp. 197–218, 1994.
- [22] C. Cifuentes and K. J. Gough, "Decompilation of binary programs," *Softw. Pract. Exper.*, vol. 25, no. 7, pp. 811–829, 1995.
- [23] L. C. Harris and B. P. Miller, "Practical analysis of stripped binary code," SIGARCH Comput. Archit. News, vol. 33, no. 5, pp. 63–68, 2005.
- [24] G. Balakrishnan and T. Reps, "Analyzing memory accesses in x86 executables," in *Proc. Compiler Construction (LNCS 2985)*, pp. 5–23, Springer Verlag, April 2004.
- [25] E. Yardımcı, C. Fensch, N. Dalton, and M. Franz, "Azure: A virtual machine for improving execution of sequential programs on throughput-oriented explicitly parallel processors," in 11th International Workshop on Compilers for Parallel Computers, July 2004.
- [26] A. Srivastava and A. Eustace, "Atom: a system for building customized program analysis tools," SIGPLAN Not., vol. 39, no. 4, pp. 528–539, 2004.
- [27] B. Buck and J. K. Hollingsworth, "An API for runtime code patching," *The International Journal of High Performance Computing Applications*, vol. 14, pp. 317–329, Winter 2000.