

Mostly-Static Program Partitioning of Binary Executables

EFE YARDIMCI and MICHAEL FRANZ

University of California, Irvine

We have built a run-time compilation system that takes unmodified sequential binaries and improves their performance on off-the-shelf multiprocessors using dynamic vectorization and loop-level parallelization techniques. Our system, Azure, is purely software based and requires no specific hardware support for speculative thread execution, yet it is able to break even in most cases, i.e., the achieved speedup exceeds the cost of run-time monitoring and compilation, often by significant amounts.

Key to this remarkable performance is an off-line preprocessing step that extracts a *mostly correct* control flow graph (CFG) from the binary program ahead of time. This statically obtained CFG is incomplete in that it may be missing some edges corresponding to computed branches. We describe how such additional control flow edges are discovered and handled at run-time, so that an incomplete static analysis never leads to an incorrect optimization result.

The availability of a *mostly correct* CFG enables us to statically partition a binary executable into single-entry multiple-exit regions and to identify potential parallelization candidates ahead of execution. Program regions that are not candidates for parallelization can thereby be excluded completely from run-time monitoring and dynamic recompilation. Azure's extremely low overhead is a direct consequence of this design.

Categories and Subject Descriptors: D.3.4 [Programming Languages]: Processors—*Incremental compilers; optimization; run-time environments*; D.1.3 [Concurrent Programming]: Parallel programming

General Terms: Design, Experimentation, Performance

Additional Key Words and Phrases: Continuous Compilation and Optimization, Dynamic Parallelization, Binary Translation

1. INTRODUCTION

As microprocessor manufacturers find it increasingly difficult to further increase clock frequencies, they are looking to parallelism as the solution that will bring the next major performance increases. Already the first processors incorporating chip multiprocessing (CMP) and simultaneous multithreading (SMT) have appeared in the marketplace. But while these designs provide a welcome performance boost for some specifically designed applications, a vast amount of legacy code cannot immediately profit from these new hardware capabilities.

In the longer term, software developers will hopefully master the use of explicit parallelism to make optimal use of CMP and SMT processors. But in the medium term, a vast number of off-the-shelf multiprocessors will be significantly underutilized for lack of software that harnesses their power effectively. Many of these future processors will include combinations of CMP and SMP, providing m parallel cores, each with n simultaneous processing contexts. The large expected variability (with respect to their parallel capabilities) of these soon-to-appear processors will greatly exacerbate the gap between potential and actual performance, reducing

the value of “binary compatibility” within processor families and potentially necessitating a different executable for every distinct internal processor configuration. The problem is compounded by a growing inventory of legacy programs that are available only in binary executable form.

Moreover, the optimal mapping of an application program to a parallel processor (n-way SMT on an m-way CMP) may actually depend on run-time inputs to the program rather than any statically extractable properties of the program itself. For example, tiling a loop among several parallel threads becomes profitable only if the iteration count exceeds a certain threshold (and the threshold in turn depends significantly on the layout of the cache hierarchy). Static analysis may determine that the iteration count is governed by a variable, whereas a run-time monitoring system could determine that the variable rarely changes during program execution. Hence, a run-time compilation system affords many additional optimization opportunities not available to traditional ahead-of-time compilers.

We have begun to address this situation by developing a lightweight run-time monitoring and compilation system that provides control speculation and loop distribution across several threads, and in which these specific optimization decisions are deferred until execution time. The goal of our system, called *Azure*, is to capture medium-grained parallelism between the lower threshold of instruction-level parallelism (nowadays handled in hardware) and the upper threshold of (grid-level) distributed computing. Unlike projects that stipulate new processor features to support thread-level speculation, *Azure* is a solution implemented entirely in software, targeting CMP/SMT hardware designs that are already commercially available, and using unmodified binaries without symbolic information as its input.

Our relatively modest research prototype of the *Azure* system is able to “break even” most of the time and repay the additional cost of run-time monitoring and optimization through a higher (and sometimes substantially improved) eventual execution performance. This is accomplished by off-loading a large part of the optimization workload to a pre-execution analysis phase that needs to be run only once for every binary input program. The analysis extracts a *mostly correct* control-flow graph (CFG) from each input program—“mostly” correct because some edges corresponding to computed branches may be missing. Instead of adding all possible CFG edges resulting from such branches at analysis time, we provide an efficient method for correctly recovering from such situations at run-time.

The availability of a *mostly correct* CFG enables us to statically partition a binary executable into single-entry multiple-exit regions and to identify those regions ahead of execution that are potential parallelization candidates. Program regions that are not candidates for parallelization can thereby be excluded completely from run-time monitoring and dynamic recompilation. As a result, only relatively few optimization candidates remain at run-time. *Azure*’s low overhead is a direct consequence of this design.

The main contributions of our work, detailed in this paper, are the following insights:

- A “mostly correct” control-flow graph as can be extracted “off-line” using static analysis provides sufficient coverage to be useful for a subsequent “on-line” dynamic optimization phase.

- Control-flow edges that are missing from such an incomplete graph can be discovered and handled efficiently at run-time.
- Static partitioning of the program into single-entry multiple-exit regions based on a “mostly correct” CFG is effective and useful and permits to focus on a very small subset of optimization candidates during the actual run-time.

The remainder of this paper is structured as follows: First, we present related work (Section 2). An overview of the Azure system follows in Section 3. Section 4 provides a discussion of our static analysis, followed in Section 5 by an explanation of the instrumentation and run-time support required to implement region-based execution of binary executables. Section 6 provides measurements and discusses these results. Notably, the benchmarks presented in this paper have all been obtained on real hardware without the use of simulation. Finally, we present our conclusions and an outlook to future research.

2. RELATED WORK

Our project touches on two substantial bodies of related work: on software-based dynamic compilation at run-time, and on techniques for exploiting mid-level parallelism at a granularity between instruction-level parallelism (ILP) and cluster computing.

2.1 Software-Based Dynamic Compilation

Several previous projects have explored extending a hardware platform by a small software layer providing run-time monitoring and/or compilation services—one also talks of a Software Extended Architecture (SEA).

- The Dynamo system [Bala et al. 2000] 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 use the optimized copy in the software cache, which over time holds all the frequently executed code.
- Kistler and the second author [Kistler and Franz 2001; 2003] implemented a system that first monitors the usage of instance variables within dynamically allocated (heap) objects, and then carries out run-time object layout changes. The system places all object fields with high access frequencies into the same cache line, improving cache performance. A second, more aggressive optimization orders the fields to reflect the cache refill mechanism.
- The ADAPT system [Voss and Eigenmann 2000; 2001] demonstrated that large performance gains are possible even when using traditional static compilers at run-time. Using a system that involved calling standard static compilers on remote machines during program execution with different optimization flags and strategies, they found performance improvements of up to 70%.
- The DyC system [Grant et al. 1999] introduced “selective dynamic compilation” to transform parts of applications at run-time, using run-time information. The run-time optimizations employed by DyC involve creating specialized versions of code regions that reflect quasi-constant values of variables. Using profiling stages

to identify optimization targets, templates are created for each target region that are then used to emit optimized code at run-time for given static variable values.

- The Master/Slave Speculative Parallelization execution paradigm [Zilles and Sohi 2001] 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. Performance increases are obtained over the original program because the distilled version has no correctness constraints.

Another advantage offered by software extended architectures is the ability to execute non-native binaries without significant loss of performance, as demonstrated by several research and commercial projects. This is useful for preserving binary backward compatibility when moving to a new platform.

- The DAISY just-in-time compiler [Ebcioğlu and Altman 1997] translates PowerPC binaries into a form acceptable to a VLIW processor [Ebcioğlu et al. 1998]. Being able to translate a non-VLIW architecture to a VLIW machine preserves the simplicity benefits of the VLIW design while overcoming the constraints of constrictive instruction set architectures.
- The Crusoe processor [Klaiber 2000] is an example for both continuous run-time 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.
- FX!32 [Chernoff et al. 1998] is a similar system that allows x86 Win32 programs to execute on Digital Alpha/NT systems. It combines an emulator with a profile-driven background optimizer that generates highly optimized code off-line, benefiting future executions of the same code.
- IA-32 EL [Baraz et al. 2003] 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 run-time “hot code optimization” stages.

Several recent projects focus on software-based techniques that perform dynamic speculative execution [Rauchwerger and Padua 1999; Cintra and Llanos 2003; Quinones et al. 2005]. These require source-code access and generally rely on profiling stages or manual manipulation of application code for parallelization. In contrast, our system takes binary executables (without any debugging information) as inputs and performs optimizations fully automatically.

2.2 Architectural Features for Medium-Grained Parallelism

Our solution targets a mid-level parallelism between ILP and cluster computing. Many researchers are exploring parallelism at this granularity. In general, one distinguishes between *explicitly parallel* solutions that expose the parallelism towards the programmer/compiler and *implicitly parallel* ones that preserve the illusion of

sequential execution. The former have a more complex programming model, while the latter require more sophisticated hardware.

Explicitly Parallel Designs. By duplicating context resources for individual threads in hardware, *Multithreaded* processors [Byrd and Holliday 1995] eliminate context-switching overhead. *Simultaneous Multithreading (SMT)* processors [Tullsen et al. 1995; Lo 1998] allow instructions from multiple threads to be issued simultaneously and thereby combine the concepts of superscalar and multithreaded execution. Chip-Multiprocessors (CMP) [Olukotun et al. 1996] are relatively simple superscalar cores independently residing on the same chip. All of these explicitly parallel hardware designs are now available in commercial off-the-shelf microprocessors.

Implicitly Parallel Designs. Trading a simpler programming model for increased hardware complexity, some highly interesting new hardware designs have been proposed to support implicit medium-grained parallelism. None of these designs have actually been built so far; research results have been obtained through simulation.

- Supertreading [Tsai and Yew 1996; Tsai et al. 1999] is a speculative parallelization technique in which hardware dependence-checking and value-forwarding mechanisms allow the static compiler to be less conservative in designating concurrent threads. Successive threads may be spawned with or without knowing that the next iteration will be executed. Data dependences between iterations are handled by hardware under explicit thread control inserted by the compiler.
- Trace processors [Rotenberg et al. 1997] build on the *trace cache* idea, storing frequently executed 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. Misspeculations due to false live-in predictions are checked by comparing the predictions with the actual live-in values at the ends of traces. Data dependencies are handled by tagging all loads and stores with sequence numbers. Partial contents of the execution state must be saved at trace entries to facilitate correct recovery.
- Multiscalar Processors [Sohi et al. 1998] partition the CFG of a program into *tasks* and attempt to execute these tasks at different parallel units with aggressive speculation. Multiscalar architectures introduce data speculation on top of control speculation; values produced by a task are forwarded to successor tasks even before validation, thus, a misspeculation in a task forces all successor tasks to also be squashed. Multiscalar processors are also quite complex; task creation and execution are controlled by a hardware sequencer, and an additional hardware buffer is needed to hold speculated memory operations and for checking dependences. Compiler support is required to partition the CFG into tasks.
- Dynamic Multithreading (DMT) Processors [Akkary and Driscoll 1998] allow threads to be created by the hardware at loop boundaries, permitting out-of-order thread execution, yet keeping track of the hierarchy between successive threads. This raises the issue of handling the greatly expanded instruction window size. DMT has a two-level instruction window mechanism in which the second buffer, referred to as the *trace buffer* is used to store speculative instructions and data. In case of a data misprediction, the instructions are fetched again, but this time

from the trace buffer. Thus DMTs have a fast recovery mechanism to offset the increased number of data dependences resulting from lack of compiler assistance. At thread entry points, DMT relies on value prediction and dataflow prediction. Data dependence checking is implemented through the usage of small, fully associative load and store queues. Issued stores are checked against the contents of the load queues in later threads, resulting in recovery operations if there is a match. All of this results in significant hardware complexity.

- Krishnan and Torrellas have proposed a Chip-Multiprocessor Architecture with Speculative Multithreading [Krishnan and Torrellas 1998; Krishnan 1998; Krishnan and Torrellas 1999]. Their approach is similar to ours in that they accept sequential binaries as their input, and run it through a static analyzer to mark entry and exit points of loop iterations. They then use hardware support to execute such loops speculatively, using a directory-based scheme to handle inter-thread data dependences.

Just like these proposals, Azure implements an implicitly parallel design. Unlike any of them, however, Azure is deployed on top of off-the-shelf explicitly parallel processors and does not require any novel hardware mechanisms.

3. OVERVIEW OF THE AZURE SYSTEM

The Azure system takes binary executables for the PowerPC platform as its input, and runs them on PowerPC-based multiprocessor systems. While doing so, Azure employs off-line static analysis and on-line dynamic code generation with the goal of making optimal use of available parallel resources.

Azure follows the path of previous “Dynamo-style” binary translation systems, using unmodified binaries (without any symbolic information) as its input. As Leung et al. [Leung and George 1999] have shown and our work confirms, deconstructing, reoptimizing and reconstructing an already optimized binary can still yield performance increases. The advantage of using unmodified binary code as input is that no new programming model is needed, and that the system is completely source language agnostic.

The central contribution of our work is the way in which we have partitioned part of the dynamic compilation task into an off-line step that is performed ahead of time only once for every binary input program, and an on-line step that occurs concurrently with every execution. Major decisions concerning the parallelizability of code regions are made ahead of time during the static analysis phase. The emphasis of this paper is on the off-line static analysis step and the preparation steps for runtime optimization.

As a fundamental design choice for the specific research questions we were investigating, no optimized code is generated during the off-line preprocessing step; only instrumentation code is inserted. All code generation for parallelization and vectorization, along with the final mapping to execution units, occurs only at run-time and strongly depends on the specific capabilities of the actual execution hardware. It would be possible to shift even more of the optimization workload to the off-line preprocessor by having it generate optimized code templates ahead of time (“staged compilation” [Grant et al. 1999; Chambers 2002]); we deliberately chose not to pursue this approach. Azure is light-weight enough that it can “break even” on many

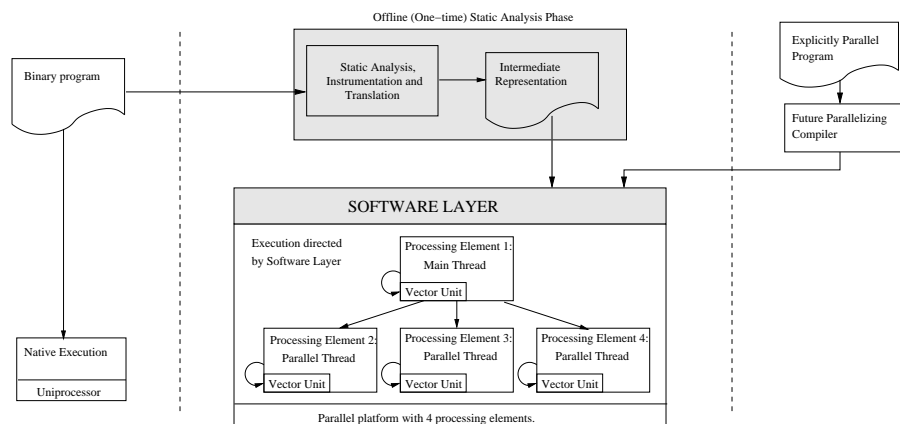


Fig. 1. Azure system is represented by shaded areas of the diagram. Binary program can either be executed directly on native hardware (at left) or run through the Azure off-line analyzer (middle) and then be executed on the dynamically parallelizing software layer. Future compilers (at right) might target the Azure intermediate representation directly.

benchmarks, in spite of performing optimizing compilation at run-time.

Figure 1 depicts an overview of where Azure fits into the larger scheme of things. The binary programs that Azure accepts as its inputs can be executed directly on any PowerPC processor (left side of figure). The Azure system is represented by the two shaded regions in the middle of the figure: an off-line static analysis component and a thin software layer that runs on top of multiprocessing hardware. The static analyzer converts the PowerPC input programs into an intermediate representation ahead of time; it is this intermediate representation that is actually executed by the Azure software layer. Finally, future parallelizing compilers might compile directly into the Azure intermediate representation (right side of figure), directly translating programmer directives expressing explicit parallelism.

The task of efficiently parallelizing pre-existing (sequential) programs therefore devolves into three sub-problems: converting a binary executable into a form that is easy to monitor and recompile, the identification of optimizable regions, and the recompilation of identified regions for vectorization and/or parallelization. The remainder of this section addresses these and other important facets of Azure’s overall design.

3.1 Directly Executable Intermediate Representation / Single Address Space

Most “Dynamo-style” dynamic compilation systems start off by using emulation to execute an input program instruction by instruction, and then periodically forward often-executed program parts to a dynamic compiler. Compiled program pieces are kept in a code cache for subsequent invocations.

Azure differs from this approach because only a small fraction of a program’s code is ever considered for parallelization—and this is mostly determined ahead of time during the static analysis phase. Executing programs via emulation would bring with it a substantial performance penalty that is unwarranted in the specific context in which Azure operates.

Taking this into account, we have designed the Azure “intermediate representation” to actually contain a native executable program, in which the non-parallelizable parts are executed normally and directly by the underlying hardware. Only the parts that are potential parallelization candidates differ significantly from the corresponding parts of the original input program, in that they have been re-written to include instrumentation code and control transfers to the dynamic compilation system. The directly executable code in the intermediate representation is augmented by extensive tables capturing the results of the static analyses.

As a further design decision, we decided to run the Azure dynamic compilation system in the same process as the program being optimized. Control is transferred back and forth between the input program and the dynamic optimizer through simple branch instructions at prescribed points that have been inserted by the off-line preprocessor. Having the application share the same process with the Azure run-time, along with its address space, does have a few drawbacks but these are amortized by several considerable advantages. The main drawback is additional load-time complexity.

The fact that the application code (belonging to the main thread) and the system code will never be active concurrently may seem at first glance to be detrimental on performance but the actual effects are negligible. The aim of the system is to obtain an optimally parallelized version of the application early on during execution, and to reduce interference to a minimum for the remaining time; this is very close to what happens in practice.

The feasibility of this approach is dependent on the validity of the assumption that the behavior of loop regions observed during the early stages of execution will be representative of their behavior in subsequent stages. This approach does not conflict with the presence of *phases* within an execution, in that code regions may be utilized with varying frequencies at different phases; we are simply maintaining that when these code regions *are* executed, they will display similar characteristics as they did previously.

As will be further elaborated on below, the fact that the processed program in the Azure “immediate representation” occupies the exact same address range as the original binary program being optimized is utilized by our system to detect additional control flow edges that the static analyzer may have missed.

In its current state, Azure accepts statically compiled 32-bit Linux/PowerPC binaries (PPC32) as inputs. In our benchmarks, we have used binaries compiled with peak performance optimization flags as directed by the SPEC organization. The non-SPEC programs were compiled with similar optimization flags. No symbol table information is extracted from the binary file; it does not make a difference whether the binaries have been stripped (using the Unix `strip` tool) or not. Extending Azure to accept binaries compiled without the `-static` flag (where library codes are attached to the executable file at load time) would be an engineering task without major additional research insights.

3.2 Loop-Level Speculative Parallelization

Azure’s parallelization strategy is geared towards splitting a loop’s iteration space and distributing the resulting tasks to available processing elements. This has an

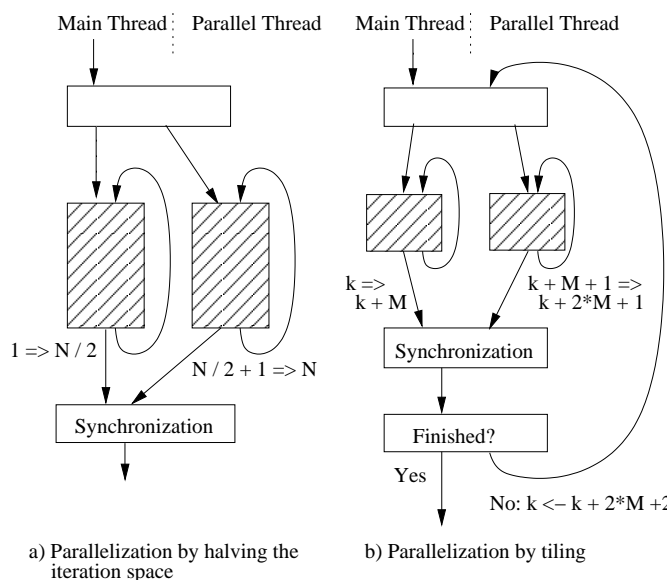


Fig. 2. Two parallelization methods. Method (a) is preferred and implemented when the iteration count is known upon entering the loop. Method (b) uses a form of loop tiling when the iteration count cannot be determined in advance.

advantage over the alternative strategy of distributing consecutive iterations across threads, in that synchronization is required only when the parallel threads have completed execution of all loop iterations.

Although this strategy results in lower overheads, it requires being able to know how many iterations a loop is going to execute. For loop bodies containing break statements, the loop iteration count cannot be predicted ahead of time, so parallelization proceeds by “tiling” the loop into fixed-size chunks (Figure 2).

Tiling of a loop introduces additional synchronization phases, in which 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.

In the Azure system, the only speculation that is carried out is based on control flow; there is currently no speculation on data. Control speculation is based on the expectation 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 [de Alba and Kaeli 2001] (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 main criteria for a loop to be considered parallelizable include the following: a) all induction registers have determinate fixed strides, b) stores to memory and induction registers post-dominate the beginning of the loop body, and c) no potential loop carries data dependences. While these may seem to constrain the number of parallelizable regions that can be discovered, we will show below that the resulting coverage often still represents a large fraction of the program in terms of execution time, thus enabling significant performance increases.

Nevertheless, relying on optimization gains only from this approach would result in passing over many loops with smaller iteration counts, which could potentially form a large percentage of executed code. Examples include nested loops in which the inner loop has a much smaller iteration count than the outer loop. Our system would prefer to have the larger iteration at the innermost loop—directly opposite to what compiler optimizations try to accomplish in loop interchange operations when optimizing for better cache utilization. Other examples include procedures consisting of small, tight loops that are called highly frequently during the execution of an application.

This problem can be overcome in different ways. One would be by utilizing a more powerful run-time loop identification mechanism that would identify nested loops and attempt parallelization by targeting the outer loop. Another is by parallelizing the inner loop using other means, such as by using vector instructions that are found in most modern microprocessors (Motorola’s AltiVec [Nguyen and John 1999], Intel’s MMX/SSE/SSE2 [Kagan et al. 1997; Thakkar and Huff 1999; Skoglund and Felsberg 2005] and AMD’s 3DNow! [Oberman et al. 1999]). The Azure system employs dynamic vectorization for exactly this purpose, as a synergistic compilation strategy in concert with dynamic parallelization.

3.3 Mapping of Tasks to Processing Elements

Based on run-time monitoring, Azure 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 exceeds a certain number, the *parallelization threshold*, the loop is admitted to the next stage, where further safety checks are made before a decision is reached on whether the loop will be recompiled or not.

The parallelization threshold can depend on many factors. We envision an installation-time decision to be made when an Azure-like system is first installed on a hardware platform. On platforms where there is close proximity between processing elements, the threshold will be lower, and on hierarchical platforms, having processing elements of different proximities (such as an n-way SMT on an m-way CMP), there will be multiple thresholds [Yardımcı and Franz 2006].

The thresholds we have utilized while developing the Azure prototype were obtained through empirical observations carried out by executing benchmarks with command-line tunable loop bodies. To obtain initial threshold values for our target platforms we used the `treeadd` and `em3d` benchmarks from the Olden benchmark suite [Carlisle et al. 1994]. These benchmarks have two convenient properties: the executions of both applications are dominated by single loop regions, and these loop counts can be controlled by changing command line arguments.

We established through experimentation that an iteration count of 2000 was a good threshold for a dual PowerPC G4 multiprocessor platform connected through main memory. For a quad POWER5 platform with both SMT *and* CMP (shared L2-cache) processors, we settled on thresholds of 100 and 500, respectively. In consequence, a loop that would fall 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 few communication requirements that it is more efficient to map parallel tasks to different chips, where each task can fully use the available cache.

Figures 3 and 4 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 system, with each core having 2 symmetric multiprocessing contexts.

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` as the loop size could not be increased beyond a certain size. 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 associated 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, at CMP-level mapping).

These thresholds only apply to the dynamic parallelization of loops. A loop with an iteration count below 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 of these optimizations (possibly because of a low iteration count or irregular strides in memory accesses) it is “released” from the active management of the Azure system and never profiled again.

4. AHEAD-OF-TIME STATIC ANALYSIS

Azure provides a one-time (per input program) static analysis stage that takes as input a statically compiled binary and emits an intermediate representation. This IR consists of a combination of executable code and various structures that enable run-time profiling and dynamic recompilation. For the global program, we provide a table that handles execution of indirect branches (explained in detail in Section 5.3), an optimization-region control flow graph, and stubs and jump tables that handle transfer of control flow between the application code and the Azure system code. For each optimization region, we provide register definition-

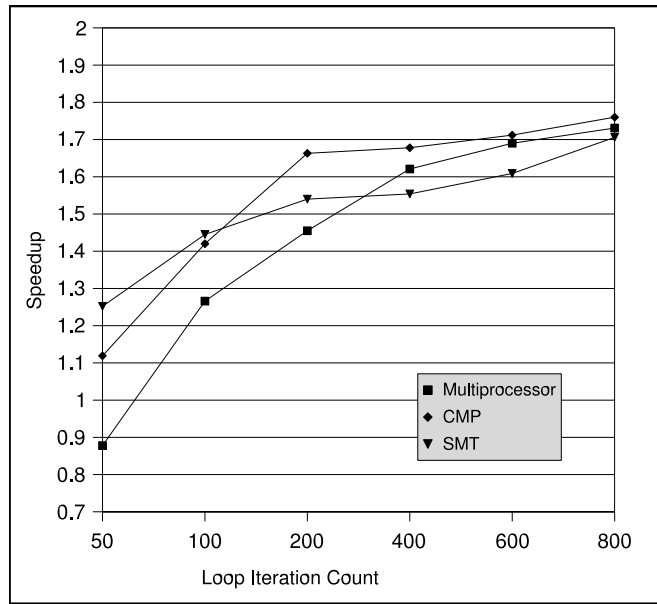


Fig. 3. Em3d speedups for various loops sizes on MP, CMP and SMT platforms.

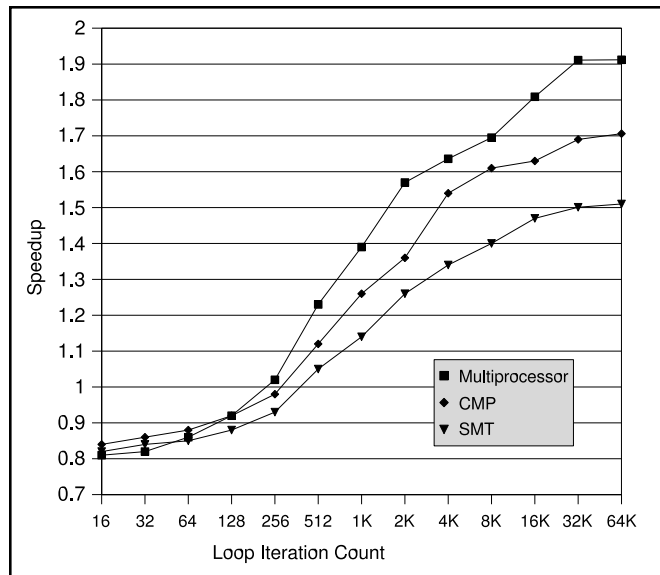


Fig. 4. Treadd speedups for various loops sizes on MP, CMP and SMT platforms.

usage information to enable efficient profiling and fast legality verification of code transformations, and advanced control flow structures such as local dominator trees to assist in the verification process.

As the static analysis involves considering the definitions generated and killed by each specific instruction in a static binary, along with the “mostly correct” control flow graph extracted from the binary code, the creation of the IR requires significant execution time and memory. The time spent for static analysis is amortized over time as the IR is created only once for a given application and can be employed again and again with different inputs.

This does require that the generated IR be saved to a file to be used in future executions of the application. This file will always be valid for the binary executable; a replication of the static analysis phase will only be necessary if the application is recompiled. The IR file and the binary executable could of course be joined into a fat binary to simplify maintenance.

4.1 Granularity of Optimization Regions

The optimization-region granularity on which our system implements optimizations is the single-entry multiple-exit *superblock* structure [Hwu et al. 1993]). The original superblock scheduling technique introduced in the 1993 paper was intended as an across-basic block scheduling mechanism that eliminates some of the complexities associated with trace scheduling [Fisher 1981] (or, for that matter, percolation scheduling).

A superblock is a collection of basic blocks into which control flow can enter from only one block but may exit from multiple basic blocks, and which does not contain any internal loops. Complexities in trace scheduling mainly arise from the fact that “hot” traces selected for optimization may include code that has control flow edges from outside the trace to locations in the trace other than the trace entry point. Maintaining correct execution for all paths then poses a problem when instructions are reordered; compensation code has to be inserted to cover all potential control flow transfers, hoping that the actual execution is skewed towards the path represented by the trace itself.

In Azure, the superblock structures are used not specifically with this goal in mind as the system does not carry out any code reordering unrelated to parallelization and vectorization, although in principle a system like Azure may also utilize this property of superblocks to carry our code reordering on frequently executed blocks that cannot be parallelized or vectorized.

Examples of possible superblock structures are shown in Figure 5, as groups of basic blocks. The superblocks are shown as the basic blocks enclosed within the ellipses. Part (a) of Figure 5 shows how two basic blocks are prevented from being grouped into a single block due to the control flow edge to basic block *B*. Part (b) depicts the boundaries of superblock expansion, when control flow edges reach basic blocks that are reached from other superblocks, and in part (c) basic block *I* is prevented from forming a superblock with basic block *H* (even though basic block *I* is dominated by basic block *H*) because the loop edge from basic block *H* to itself stops any expansion of a superblock boundary beyond basic block *H* itself.

The benefits of having a superblock-level execution granularity for our system is twofold. First, there is a reduction in profiling overhead. Instead of instrumenting

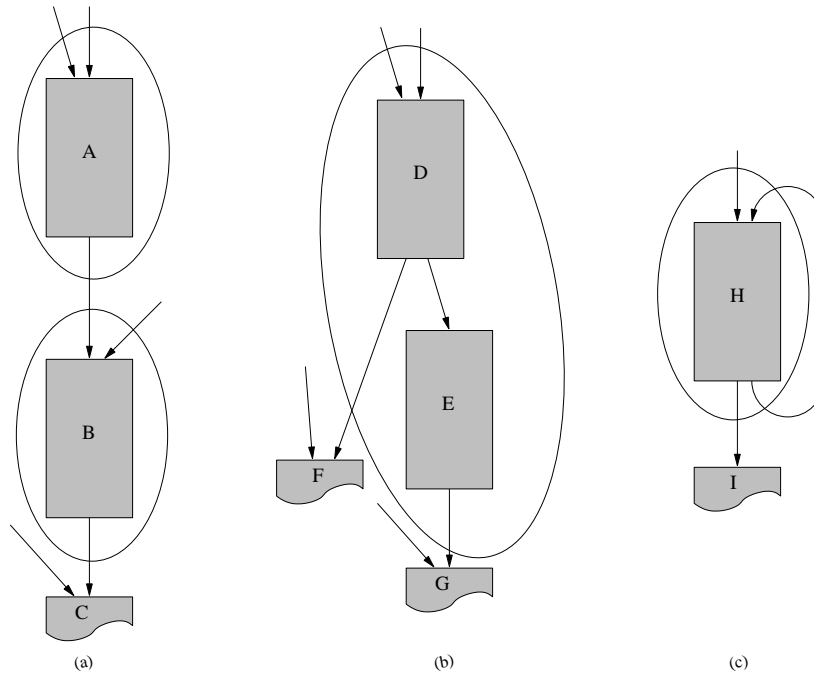


Fig. 5. Examples of superblock structures, displayed as groups of basic blocks within ellipses. Each shaded rectangle is a basic block of instructions, where upon entry execution is guaranteed to proceed to the last instruction. Parts (a) and (b) depict how superblock limits are defined by the single-entry paradigm, and part (c) shows the limiting effect of loops.

each basic block to obtain run-time profile information, only the basic block that dominates a group of basic blocks (not going beyond loop boundaries) is instrumented. Second, having superblocks as the building structures for parallelization regions (as opposed to loops) enables us to optimize complex loop bodies without excessive code growth. Note that while the granularity for monitoring and recompilation is at superblock level, an actual loop that has been identified as a target for optimization may consist of multiple superblocks.

Although our system does not carry out other kinds of run-time optimizations such as strength reduction, code straightening or loop unrolling, these optimizations would also benefit from the superblock-based granularity of our system.

4.2 Extracting Control Flow

A major step in creating the Azure Intermediate Representation is the partitioning of a binary executable into single-entry multiple-exit superblock regions. This occurs in two steps, in which first a Control Flow Graph (CFG) is extracted from the executable, partitioning it into basic blocks. Then, basic blocks are combined to obtain non-overlapping superblock regions of the largest possible size.

The single biggest problem in this step is the presence of indirect branches, which jump not to a fixed offset from the branch's program address (depending on the existence and outcome of conditional expressions), but to an address contained

within a CPU register. For the remainder of this document, all other types of branches will be broadly referred to as “direct” branches.

This is a problem for two reasons: First, because the dynamic monitoring mechanism of the Azure system requires insertion of instrumentation code before superblock entry points, and the resulting motion of code segments with respect to each other requires updating of branch instructions. While branches that have jump offsets embedded within the instruction format may be statically updated to reflect the new locations of target addresses, indirect branches cannot all be modified this way.

Although *call-return* instructions pairs (implemented through the `bl-blr` instructions in PowerPC (explained in Figure 6) are safe because in these situations the new (correct) target address is set by the call instruction itself, other uses of indirect branches involve loading a statically declared address (or even offset) from memory and jumping to the loaded (or calculated) address, and these break the legality of arbitrary post-link code motion. Even call-return pairs may be corrupted if any arithmetic is done on the target address values and stored back to the relevant indirect branch register between the call and return instructions.

The second, more general reason why indirect branches cause problems is that the superblock-granularity of our system requires very precise control-flow information. By definition, control should enter a superblock at a single location, namely the entry point. An initial, “naive” pass through the instructions gives us a general structure of the CFG by incorporating edges originating from direct branches. However, there may be indirect branch edges that cannot be detected by scanning the instructions statically, and some of these may target a location that was *not* determined to be a superblock entry point in the naive CFG-generation pass.

Our solution to this problem is to use a “mostly correct” CFG obtained during static analysis along with a low overhead detection mechanism at run-time for the occasional additional control-flow edge that the static analysis may have missed. If such an additional edge is detected that violates the statically determined superblock structure, then the existing optimization decisions on the superblock(s) in question must be invalidated from this point onward.

Figure 6 enumerates the instructions in the PowerPC/POWER architecture that either read or write the special-purpose registers that may hold indirect-branch target addresses. These registers are the *Link Register* (LR) and the *Count Register* (CTR). All indirect branch instructions by definition read the LR and CTR (except for the `blr1` instruction which reads *and* writes to the LR at the same time), and dedicated “move-to-special-purpose-register” (`mtlr`, `mtctr`) and “move-from-special-purpose-register” (`mflr`, `mfctr`) instructions write and read, respectively, to these registers.

The write instructions that are listed in Figure 6 are the ones that write to these registers for the purpose of indirect branching; certain conditional branch instructions (such as the `bdnz` branch), which are not listed, decrement the CTR contents and evaluate the condition depending on whether CTR is zero or not (along with the embedded conditions of the branch itself). This is due to the dual nature of the CTR register in the PowerPC/POWER instruction set architecture; this register can be used to implement indirect branching as well as loop counters.

Instruction	Functional Description	Sets LR	Sets CTR	Uses LR	Uses CTR
blr	Jumps to address in LR	No	No	Yes	No
blrl	Jumps to address in LR	Yes [†]	No	Yes	No
bctr	Jumps to address in CTR	No	No	No	Yes
bctrl	Jumps to address in CTR	Yes [†]	No	No	Yes
bl <X>	Jumps to address <X>	Yes [†]	No	No	No
mtctr rN	Copies register rN to CTR	No	Yes	No	No
mtlr rN	Copies register rN to LR	Yes	No	No	No
mfctr rN	Copies CTR to register rN	No	No	No	Yes
mflr rN	Copies LR to register rN	No	No	Yes	No

Fig. 6. The list of POWER/PowerPC instructions that read or write the contents of the CTR and LR registers, for the purpose of our analyses. This list does not include conditional branches of the form `bdnz`, which decrement CTR and evaluate to true or false depending on whether CTR is zero or not. These instructions use CTR as a counter and not as an indirect branch target. [†]These instructions set LR to the address of the next instruction.

All of the branch instructions that implicitly set LR, displayed in columns 2, 4 and 5 of Figure 6 have conditional versions as well. These are essentially the same, except that the LR register is only updated if the branch condition evaluation results in the branch being taken. This, in turn, results in modest changes in how certain instructions are translated in order to maintain validity of instrumented code.

4.3 “Mostly Correct” Control Flow Information

Having described the sources of difficulties in obtaining perfect control flow graphs, the question that should now be answered is how much completeness and correctness do we actually require from the control flow information in order for it to support a useful IR structure.

The answer is that although perfect information would be highly desired, it is neither a necessity nor an always attainable target. The latter is due to complex pointer arithmetic that is not often encountered in actual programs, but which can still exist in legal binary code. Some types of arithmetic *can* be analyzed with time-consuming analysis stages, but others can never be resolved statically—such as those using user input to determine offsets to jump tables.

As our results show, perfect control flow information is not a necessity. As long as we can implement mechanisms to handle undetected control-flow edges (that violated the single-entry property of the superblock structure), and can ensure that no *live* registers are ever mistakenly designated as *killed*, our system can preserve correctness of execution.

However, obtaining good superblock structure information *is* important in carrying out effective register def-use and liveness analyses, and we want the statically-created superblock structures to result in a minimal number of run-time “side entries”, that is, control flow transfers to a location other than a designated superblock entry. Such entries are handled by our system in a lightweight manner, but they would still form a large overhead if they occurred at any significant frequency. Having detailed register def-use and liveness analysis is crucial in determining whether

a code region has enough dead registers to support lightweight profiling—saving and restoring register values to support instrumentation code would introduce unnecessary overheads. Therefore we need to information about indirect branch edges even when those edges may not necessarily belong to a parallelizable region.

As a result, it is clear that what is required is a mechanism to obtain as good a CFG as possible from the binary in reasonable time, without introducing many spurious control flow graph edges. A naive idea would have been to conservatively identify all basic blocks or even all individual instructions as targets of indirect branches. While this would eliminate unseen paths, it would also unnecessarily reduce the size of superblocks. In order to judge the merits of such mechanisms, we define the evaluation criteria as being (1) “acceptable” one-time static analysis time, depending on program size, and (2) high indirect branch coverage.

We define *coverage* in the context of our static analysis as the percentage of indirect-branch control flow edges that were actually followed during the execution of the binary program that were correctly identified during the static analysis stage. The coverage figures were obtained by executing the benchmark programs with the longest proscribed input sets, such as the *ref* input set for SPEC CPU 2000 benchmarks.

The coverage metric is divided into two subcategories. The first of these is *static coverage*, which is the coverage percentage of indirect branches based directly on the number of indirect branches. The second is what we refer to as *dynamic coverage*, which is the coverage percentage of indirect branches with each branch weighted by the number of times it was accessed during execution.

In other words, in a binary executable with N distinct indirect branch edges (a single indirect branch instruction may have more than 2 different outgoing edges), if we define:

$$n_i = \text{Number of executions of indirect branch edge } i, \text{ and}$$

$$c_i = \begin{cases} 0 & \text{Indirect branch edge } i \text{ identified by Static Analysis,} \\ 1 & \text{Indirect branch edge } i \text{ not identified.} \end{cases};$$

the coverage metrics can then be defined with the following equations:

$$\text{Static Coverage} = \frac{\sum_{i=1}^N c_i}{N}$$

$$\text{Dynamic Coverage} = \frac{\sum_{i=1}^N c_i \cdot n_i}{\sum_{i=1}^N n_i}$$

Again, although a perfect control flow graph information would be highly desired, perfect coverage is neither a necessity nor always-attainable—we explain in Section 5.3 how we deal with those special cases in which an indirect branch terminates in the middle of what was previously considered to be an indivisible superblock.

In Azure, we have implemented a heuristic that gives us useful indirect branch target information with reasonable effort, without introducing many spurious control flow graph edges.

4.4 Constructing a “Mostly Correct” CFG: The LR-Heuristic

A naive single pass does not provide enough insight into the structure of the dynamic control-flow graph, and reaching-definition iterations result in an extremely long and memory-intensive static analysis stage, as the input set consist of binary executables with possibly hundreds of thousands of separate nodes.

Our method involves starting with the imperfect CFG obtained through scanning the instructions and adding edges, performing iterative reaching definition analysis to create def-use chains, and obtaining values of target registers at indirect branch locations.

In this manner, we get all definitions made by branches that set the target registers as well as those made by explicit load instructions. Restarting the iterations after adding the newly found edges was seen to be very time- and memory-consuming, and would not result in noticeable improvements. Instead, we start a new, special iteration phase based on a heuristic that we call the “LR-Heuristic”. This phase results in near-100% dynamic coverage of indirect branches for most SPEC CPU2000 Integer benchmarks, and in zero or a negligible number of side entries to superblocks at run-time, while reducing the number of redundant edges added. Benchmarking results to assess coverage are given below.

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 are then repeated until no new edges can be found. In practice, rather than iterating to a fixed point, actual implementations often limit the number of iterations to a small number of passes that create a CFG of sufficient quality for the task at hand, such as measuring program behavior [Larus and Ball 1994], decompilation [Cifuentes and Gough 1995], or analyzing memory access [Balakrishnan and Reps 2004].

In our particular case, 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. We did initially implement such a system for Azure, but for programs having hundreds of thousands of instructions the execution costs were prohibitive. Allowing multiple iteration stages to settle down after inserting new control-flow edges at the end of each stage would take on the order of tens of hours on contemporary computing platforms. Even though this stage would only be carried out once per application, we cannot completely ignore the execution time of the static analysis stage as applications may be updated from time to time. We managed to avoid this extreme cost without a noticeable decrease in accuracy.

As previously mentioned, a naive solution, albeit completely legal in terms of correct execution, would be to conservatively identify all basic blocks as targets of

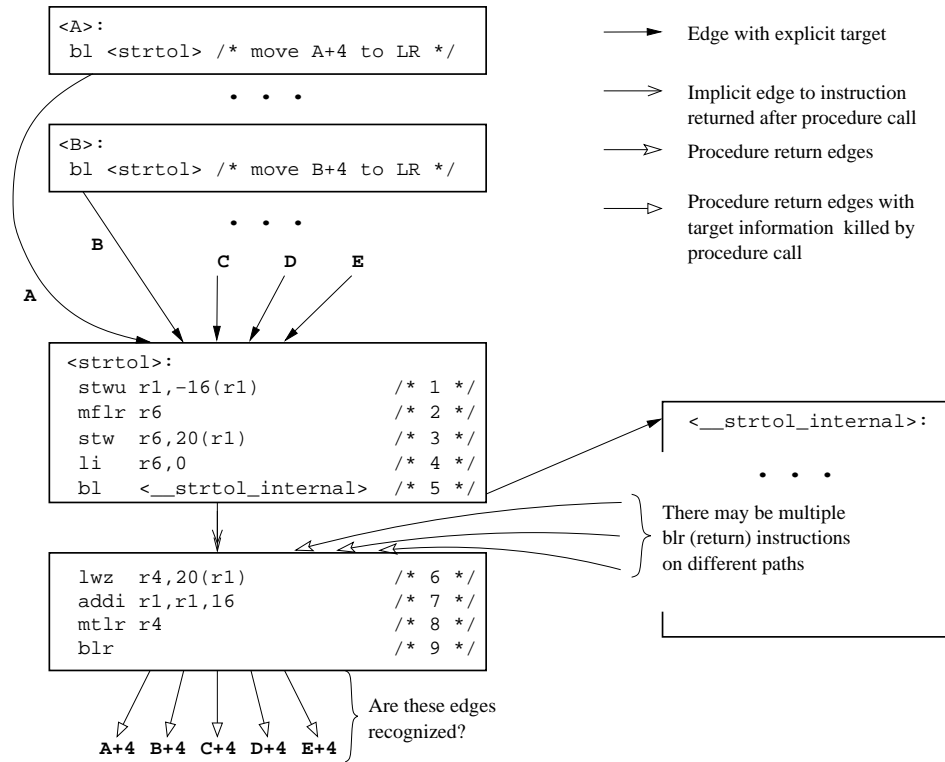


Fig. 7. Control flow edges associated with a library call to the `strtol` procedure. The `bl` instruction kills the definitions of LR reaching itself in the initial reaching definition analysis since it writes to the LR register. The LR-Heuristic is a specialized reaching definition analysis to overcome this problem and incorporates the edges at the bottom of the `strtol` code to the CFG.

indirect branches and use the resulting graph as the control flow graph¹. While this naive solution would eliminate unseen paths, it would also limit aggressive optimizations.

Our approach also starts with an imperfect CFG, and performs iterative reaching definition analysis to create def-use chains and obtain values of CTR and LR registers at indirect branch locations. Thus, we can get all LR definitions made by LR-setting `bl` instructions as well as by immediate loads to the LR and CTR registers. Restarting the iterations after adding the newly found edges was seen to be very time- and memory consuming. Furthermore, close analyses of the code showed it would not result in further improvements, as most indirect branch target addresses were being saved to memory and later restored just before the indirect branch.

¹Even though it is still possible after static analysis to have indirect branches target locations inside basic blocks, these situations are very rare. As can be seen in Figure 12, the basic block boundaries created through static analysis are almost always identical to those observed at runtime, and our side entry handler mechanism (explained in Section 5.1) maintains legality of execution for the transgressions that do occur.

Figure 7 shows a typical usage of the LR register in a PowerPC procedure and helps depict the issues faced when trying to construct a CFG from binary executables, without using symbol table or source information, nor assumptions about the specific Application Binary Interface (ABI). An ABI describes various standards on how compiled code is expected to behave, in order to make cohabitation with pre-compiled library and system codes possible. This includes the reservation of certain registers for specific functions (such as the usage of register `r1` in PowerPC as the stack pointer), and how arguments are passed. However, it is perfectly possible to produce self-contained binary code that does not observe the native ABI. For this reason, and in order to have a highly general system that can be ported to other platforms, our analyses are explicitly ABI agnostic.

The code displayed in Figure 7 is the wrapper procedure for the `strtol` Linux library procedure. Upon entry to the procedure, the contents of LR are pushed to the stack in instructions 2-3, and we can clearly see that after calling and returning from another procedure in instruction 5, the contents of the LR saved in instruction 3 are restored back to the LR in instruction 8. However, it is also possible to have code inside the procedure that does not conform to the ABI and that corrupts the LR value stored on the stack. Looking at this code sequence from a purely ABI-agnostic point of view, the static analyzer sees that the definition of LR is *killed* by instruction 5 (by writing the address of instruction 6 to LR), and some value is loaded from memory at instruction 6. The contents of the latter are obtainable only through exhaustive iterative analyses.

What is required is an ABI-agnostic analysis method that nevertheless recognizes that the value of LR stored by instructions 2-3 in Figure 7 *can* be restored by instructions 6-8. Our specific “LR-Heuristic” overcomes these problems. It results in near-100% dynamic coverage of indirect branches for most SpecCPU2000 Integer benchmarks, resulting in zero or negligible number of side entries to superblocks while minimizing the number of redundant edges added.

The classic reaching definition analysis iteratively calculates:

$$\begin{aligned} REACH_{out}(b) &= GEN(b) \cup (REACH_{in}(b) - KILL(b)) \\ REACH_{in}(b) &= \cup REACH_{out}(p), p \in predecessor(b) \end{aligned}$$

The LR-Heuristic is essentially a modified reaching definition analysis with modified definitions of *GEN* (generate) and *KILL*, which incorporates certain results obtained during the initial, standard reaching definition analysis.

In the LR-Heuristic reaching definition phase, `mflr` (move-from-link-register instructions *generate* and `blr` (branch-to-link-register) instructions *kill* definitions of LR. Notice that in the initial phase, `mflr` instructions simply *use* an existing definition and extend the reach. Thus, for each indirect branch dominated by a `mtlr` (move-to-link-register) instruction (typically residing in an earlier location in the same basic block with no intervening write to LR), we find the reaching “LR-Heuristic” definitions generated by `mflr` instructions.

The code segment in Figure 8 illustrates this mechanism. The initial reaching definition analysis allows us to see that the value of LR at 4 is `0x100000c4`, is copied to `r0` and stored into memory. At 12, a value is loaded from memory to `r0` and written into register LR.

Since the `mtlr` instruction kills any LR definition reaching it, we cannot realisti-

```

1 0x10000c0:  bl    0x10000104    // LR <- 0x10000c4, jump to 0x10000104
2 0x10000c4:  bl    0x1000020c    // LR <- 0x10000c8, jump to 0x1000020c
...
3 0x10000104: stwu  r1,-32(r1)    // Mem[r1 - 32] <- r1, r1 <- r1 - 32
4 0x10000108:  mflr  r0            // r0 <- LR
5 0x1000010c:  stw   r30,24(r1)   // Mem[r1 + 24] <- r30
6 0x10000110:  stw   r0,36(r1)   // Mem[r1 + 36] <- r0
7 0x10000114:  bl    1008c78c     // LR <- 0x10000118, jump to 0x1008c78c
8 0x10000118:  mflr  r30          // r30 <- LR
9 0x1000011c:  lwz   r0,8(r30)   // r0 <- Mem[r30 + 8]
10 0x10000120:  cmpwi r0,0        // compare r0 with 0
11 0x10000124:  beq-  0x10000130  // jump to 0x10000130 if r0 == 0
12 0x10000128:  mtlr  r0          // LR <- r0
13 0x1000012c:  blrl              // LR <- 0x10000130, jumps to previous LR

```

Fig. 8. LR-Heuristic example. In this PowerPC code segment, the LR-Heuristic identifies 0x10000c4 as a possible target of instruction 13. The semantics of the instructions that affect the LR-Heuristic method are described in the comments to the right.

cally make assumptions on the address and value of the load at 9—without making assumptions on the ABI—because the code in between jumps to different locations. Our analysis at this initial stage cannot predict the target of the indirect branch at 13. Acting on the observation that little actual arithmetic is carried out on LR registers, we implemented the simple LR-Heuristic method to obtain a more detailed analysis of indirect branch targets.

Looking again at the code segment, we can see that this method identifies the `mtlr` instruction as killing all reaching definitions of LR. With the heuristic-based iterative analysis, we are able to further identify the LR definitions by `bl` instructions at 1 and 7 as reaching the `blrl` instruction. In fact, it turns out in the end that 0x10000c4 is the target of this particular indirect branch.

As previously stated, perfect coverage is not attainable, nor is it necessary. We would like to have as correct a control flow graph as possible, but also allow for the possibility of rare side entries to structures that the static analyzer has mistakenly designated as indivisible superblocks. We handle these rare cases with what we call an *Indirect Branch Table (IBT)*, explained in Section 5.3.

4.5 Superblock Construction

Figure 9 outlines the basic algorithm to partition the control flow graph, composed of basic blocks, into superblock groupings. The algorithm itself is fairly straightforward, running in $O(n)$ execution time.

The algorithm in Figure 9 starts with a list of basic blocks not belonging to any superblock and tries to join as many basic blocks together as possible within the superblock definition criteria. The `expandSuperBlock` procedure attempts to join a basic block (let us call this the *seed* basic block) with its successors: If a successor block has no other predecessor blocks, it is incorporated into the seed block’s superblock. If a successor block does have a predecessor belonging to another superblock, it is marked as being the entry point of a new superblock. The algorithm then tries to expand this new superblock in turn through a recursive call to `expandSuperBlock`. If, however there is a predecessor not marked as belonging

Algorithm 4.1: SUPERBLOCKCONSTRUCTION(*BasicBlockList*)

```

global SuperBlockList
procedure EXPANDSUPERBLOCK(block, superblock_in)
  if superblock_in = NULL
    then { BasicBlockList.remove(outblock)
          block.superblock ← NEW(block)
          SuperBlockList.add(block.superblock)
        }
    else block.superblock ← superblock_in
  for each outblock ∈ SUCCESSOR(block)
    do if outblock.MustBeSuperBlockEntry
    then EXPANDSUPERBLOCK(outblock, NULL)
      if outblock.superblock = block.superblock
        then Continue;
        indeterminate_entry ← false
        other_superblock_entry ← false
        for each inblock ∈ PREDECESSOR(outblock)
          do if inblock.superblock = ∅
            then indeterminate_entry ← true
          else if inblock.superblock ≠ block.superblock
            then other_superblock_entry ← true
        if indeterminate_entry and not other_superblock_entry
          then MultipleEntryBasicBlockList.add(outblock)
          else if other_superblock_entry
            then EXPANDSUPERBLOCK(outblock, NULL)
          else { BasicBlockList.remove(outblock)
                EXPANDSUPERBLOCK(outblock, block.superblock)
              }
      indeterminate_entry ← false
    for each outblock ∈ MultipleEntryBasicBlockList
      do for each inblock ∈ PREDECESSOR(outblock)
        do if inblock.superblock ≠ block.superblock
          then { indeterminate_entry ← true
                break
              }
    if not indeterminate_entry
      then { BasicBlockList.remove(outblock)
            EXPANDSUPERBLOCK(outblock, block.superblock)
          }
  main
  for each block ∈ BasicBlockList
    do EXPANDSUPERBLOCK(block, NULL);

```

Fig. 9. Pseudocode describing the superblock-creation algorithm.

to any superblock, then the block is marked as possibly having multiple entries and its superblock status is resolved only after all predecessors have been identified.

Basic blocks are analyzed repeatedly until each of them belongs to a superblock, and the algorithm results in an initial list of superblocks references which may include loops within the superblock structure. The loops are removed by a second $O(n)$ stage, as the definition for our superblocks stipulates that they do not expand beyond loop boundaries. The resulting superblocks typically range from 1 to 5 basic blocks.

There is a reference within the algorithm to a query whether a basic block “must be” a superblock entry (thus preventing even those basic blocks that dominate it from being grouped into the same superblock). These are special cases, and whether a basic block *must* be a superblock starting point is decided before the partitioning algorithm.

Indirect branches using the CTR register (`bctrx` instructions) are mostly seen to be utilized for reading an address from memory, moving it to the CTR with the `mtctr` instruction and then branching to it. Thus, without access to symbol table information (something we are assuming is not available to us) analysis of indirect branches using the CTR register is not a practical task.

However, we have observed that indirect branches using the CTR register jump mainly to procedures or other pieces of code that were not reachable by branches with explicit addresses. Therefore, our superblock-creation algorithm automatically identifies these as superblock entries, preventing superblock side entries from occurring at run-time later. As these indirect branches may also set LR (the `bctrl` version), any `blr` instruction used for procedure returns would then return to the address following the `bctrl`. Thus, during initial parsing of instructions, Azure marks these addresses as “*must be*” superblock entry points for the subsequent superblock creation stages.

Another method we employ to decrease the number of potential side entries to superblocks is the traversal of the static data area of the executable binary, seeking data values that might be pointers to code addresses. The executable file format (in our case the Unix ELF format) provides us with knowledge of the location of the static data section along with the boundaries of the executable code virtual address space. With this information, we declare all basic block starting addresses that we find within the static data section as “*must be*” superblock entries.

Another type of code that we declare beforehand as “*must be*” entry points for superblocks are basic blocks ending with `blr1` instructions. In fact, for reasons explained below (directly related to the process of maintaining correct execution under the superblock-based execution model), we split such basic blocks to isolate the `blr1` instructions and declare those instructions as distinct superblocks. This does not cause a significant change in the resulting superblock list, as relatively few `blr1` instructions exist in compiled binaries, and even fewer get executed at run-time.

4.6 Program Coverage Results

In order to have an idea about how well the LR-Heuristic performs, we measured the coverage (static and dynamic) obtained using the LR-Heuristic method, and also using just a “basic” initial reaching definition analysis stage. For this analysis, we

chose the SPEC CPU2000 Integer benchmarks, representing a group of programs that have complex control flow. All benchmarks were compiled on a native PowerPC machine using GCC 3.3.2 with peak SPEC optimization flags. All results were obtained without any simulation on real hardware by running the reference input set until completion.

After performing control-flow analysis, we inserted checking instructions into the binaries to obtain the dynamic indirect branch target information. As certain indirect branches are executed much more often than others, we also instrumented the binaries to obtain dynamic execution counts. As a result, we obtained enough information to be able to comment on the correct coverage of our static analysis as a percentage of both distinct indirect branches and dynamic indirect branch executions.

Static Coverage. Figure 10 presents the static coverage results, in other words the percentage of indirect branches whose target addresses Azure’s control flow analysis was able to completely identify ahead of time. The notion of “complete” identification stems from the fact that we consider an indirect branch as “covered” only if *all* edges from that indirect branch were identified. In the figures, the black bars show the results obtained with the basic analysis and the white bars show results obtained with the additional application of the LR-Heuristic.

Giving a specific example on how the results were obtained, of the 156 indirect branches that were executed at least once while running `gzip` with the reference inputs, the basic analysis was able to find all the targets in fewer than 30% of the cases. Application of the LR-Heuristic increased this percentage figure to over 80%. Across the complete range of benchmarks only in `crafty` can the basic control flow analysis obtain a coverage figure of over 30%. The LR-Heuristic method increases this to over 80% on most benchmarks, passing 90% in two of them.

Dynamic Coverage. We are more interested in knowing the effect on the overall program. Finding the targets of many rarely executed indirect branches is less important than finding those of a select set of frequently executed branches.

To obtain this information we inserted instrumentation code to count the executions of each indirect branch; Figure 11 shows the resulting dynamic coverage data. Except for two benchmarks, the coverage increased dramatically. For 9 of the 12 benchmarks, the coverage was greater than or equal to 95% (approaching 100% in most). For `gcc` it was around 90%. In `perlbnk` and `gap`, the coverage dropped dramatically, to less than 40% dynamic coverage.

Looking further into these results, we see that a majority of the “missed” indirect branch targets result from just a few indirect branches: In `perlbnk` one missed indirect branch, a `bctrl` instruction (jumping to addresses loaded from memory) represents over 60% of all executed indirect branches. A close analysis of the targets of such branches shows an overwhelming preponderance of procedures unreachable from other instructions (except via indirect branches), with values in registers being saved by the callee. In `gap` the most frequently executed missed indirect branch is a `blr` return instruction at the end of a small procedure. This procedure is also not reachable from other locations and register values are again preserved by the callee. Superblock creation and liveness analysis are not likely to be noticeably affected by

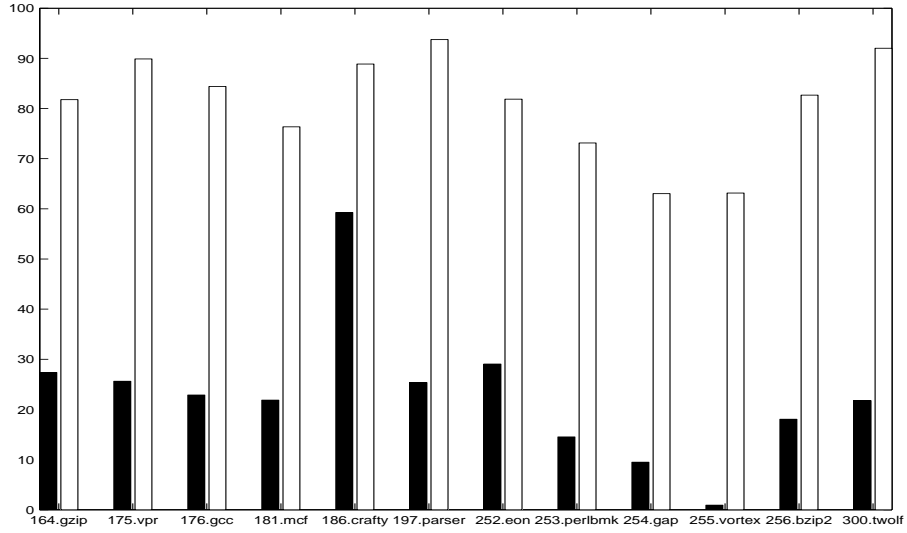


Fig. 10. Static percentages of indirect branches whose targets were completely identifiable. The black bar shows the “basic”, the white bar the LR Heuristic analysis results.

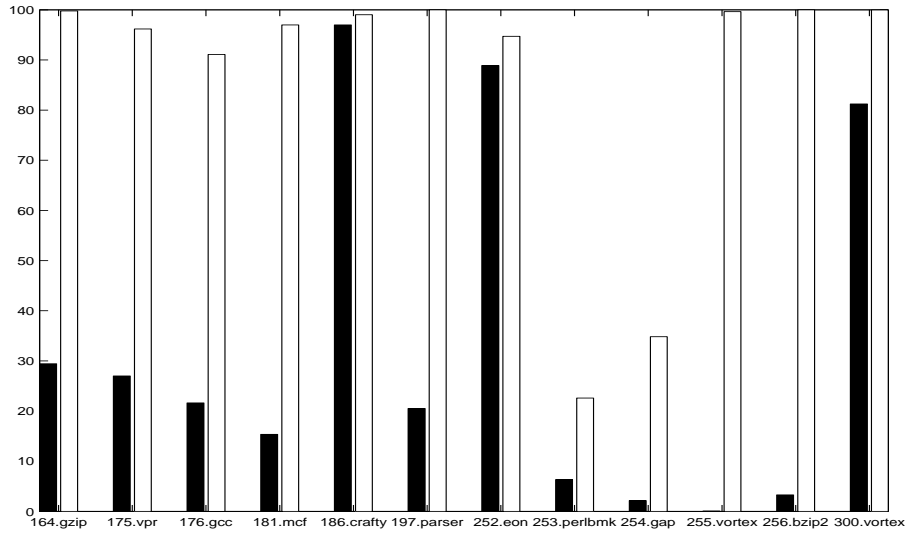


Fig. 11. Dynamic percentages of indirect branches whose targets were completely identifiable. The black bar shows the “basic”, the white bar the LR Heuristic analysis results.

Benchmark	Superblock Side Entries	Number of Indirect Branches	Unique Side Entry - Target Address Pairs
gzip	0	1.65×10^{10}	0
vpr	0	2.89×10^9	0
gcc	1205036	1.51×10^{14}	69
mcf	0	5.64×10^7	0
crafty	0	2.83×10^{11}	0
parser	1	3.16×10^{11}	1
eon	4	2.46×10^{10}	1
perlbmk	116933	2.99×10^{11}	7
gap	1	9.72×10^{10}	1
vortex	11250	1.87×10^{11}	12
bzip2	0	5.20×10^{10}	0
twolf	0	1.02×10^{10}	0

Fig. 12. The number of side entries to superblock-designated structures, the dynamic number of indirect branches, and the unique indirect branch-target address pairs that make up the side entries. Even when side entries do occur in an application, they are rare compared to the total number of indirect branches. Since all indirect branches result in a superblock control-transfer, this is actually a lower bound on the total number of superblock entries.

either such case.

Implications of Coverage Results. In order to better quantify these observations, we instrumented superblock entry and exit points to identify the number and locations of side entries to superblocks. Side entries originate from unidentified control flow edges in imprecise superblock structures, which in turn originate from indirect branch edges not discovered by the static analysis. To obtain an idea of what percentage of superblock entries are side entries, we compare against the execution counts of indirect branches. As all indirect branches are superblock exits, this is a lower bound on superblock control transfers. Figure 12 summarizes the results. As one can see, side entry points are extremely rare.

For `gcc` and `perlbmk`, the number of side entries do not seem as negligible as those in other benchmarks, but neither are they at all significant as a percentage of all superblock control transfers. Analysis of these side entries is interesting in that a few control transfers from one superblock to another causes the vast majority of such entries. In `gcc` 96.5% of superblock side entries are caused by just 5 *unique control transfers* (a unique control transfer denotes unique pairs of indirect branch location and target address). In `perlbmk` *all* irregularities are caused by just 7 unique control transfers.

While executing under Azure, such infrequent occurrences of side entries to superblocks are detected by the system and handled without causing any noticeable effect on overall performance.

Object-oriented Code. One of the SPEC CPU2000 Integer suite programs is a C++ program, `eon`. Furthermore, although written in C, `vortex` is an object-oriented database that creates and transacts with database objects. It is reasonable to expect that these applications might have different characteristics with respect to the number of indirect branches and their behavior. Looking at Figure 12, which

Benchmark	Total Number of Superblocks	Statically Identified as Feasible	Dynamic Entries to Feasible Superblocks	Parallelized Superblocks
gzip	10900	283	19.7%	9
vpr	12679	342	0.1%	0
mcf	9415	244	11.7%	5
parser	13655	300	5.8%	11
bzip2	10839	286	34.3%	10
twolf	13388	437	31.0%	7
perlbnk	14736	478	59.2%	4
art	12391	322	58.7%	12
swim	10012	383	32.2%	11
treeadd	9821	201	97.8%	1
em3d	9318	188	91.5%	1

Fig. 13. 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. The last column shows how many superblocks are actually parallelized in the end.

lists the number of static indirect branches in the input binaries, we do not see an outlier in the total number of indirect branches. This is likely due to library code in the static executable taking up most of the binary code space.

Looking at the static coverage results in Figure 10, while the coverage results of `eon` are near the average, very few of the indirect branch targets in `vortex` can be identified with the basic identification algorithm; the LR-Heuristic greatly improves coverage here. This effect is even more pronounced when we look at the dynamic coverage results (displayed in Figure 11).

While these two benchmarks are not sufficient to reach a conclusive verdict on the effect of object-oriented applications on our coverage results, it seems reasonable to expect more “troublesome” indirect branches in object-oriented code. We can also expect that the LR-Heuristic will have an even higher impact on coverage for such programs.

4.7 Static Non-Optimizable Region Elimination

After partitioning an input program into single-entry multiple-exit superblock regions, the static analyzer performs a conservative feasibility analysis during which it identifies all superblocks that can statically be determined not to be optimization candidates.

For example, because Azure focuses on loops, all superblocks that don’t form a loop with at most one other superblock are excluded from further consideration at this point, as are loops with input registers that have indeterminable strides. Only the superblocks that remain after this pruning step will actually be profiled and dynamically optimized at run-time; the superblocks that have been excluded statically will not be touched again by the Azure run-time system and will execute in the exact same form as they appeared in the original program binary.

As a result of the static elimination step, only a tiny fraction of an input program’s code is actually profiled and dynamically recompiled at run-time. Yet many pro-

Benchmark	Overhead of Considering All Superblocks at Run-Time	Overhead of Considering Only Feasible Superblocks 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%
perlbnk	9.4%	1.4%
art	8.9%	1.1%
swim	8.4%	1.2%
treeadd	3.1%	0.4%
em3d	7.1%	0.3%

Fig. 14. Static elimination of non-feasible program regions from further consideration significantly reduces dynamic system overheads.

grams spend a significant proportion of their run-time in precisely the superblocks that were statically identified as being feasible.

Figure 13 presents a very rough approximation to illustrate this point. Because it is not easy to measure the actual time spent in each superblock, we are instead counting the number of dynamic entries to superblocks. The third column in the table gives the proportion of dynamic entries to superblocks that went to those that had previously been identified as being feasible, obtained by instrumenting every single superblock in the program with a counter. This analysis completely ignores the fact that some superblock regions are much larger than others, and that some contain loops while others do not. However, because *all* feasible superblocks are associated with loops, it is likely that in many cases the proportional time spent in feasible superblocks is even higher than their proportional access frequency.

Figure 14 presents the effect on system overheads that is brought about by statically excluding all program regions from further consideration that are guaranteed not to contribute any benefit. An interesting aspect of our approach is that programs that have a low dynamic coverage of feasible superblocks as a result 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 it would have been if the Azure system had not been present in the first place.

4.8 Register Analysis

The final step in the static analysis stage is the analysis of register values, using the control flow graph augmented with the edges uncovered during the basic and LR-Heuristic stages.

Register availability information, which includes *dead* and *used* register information, is calculated at a basic block granularity and utilized at superblock-level locations. In other words, in an initial step, reaching definition and liveness analy-

```

<loopA>:
    addi r3,r3,64 // r3 <- r3 + 64
    lwz  r8,12(r3) // r8 <- Mem[r3 + 12]
    mulw r8,r8,r7 // r8 <- r8 * r7
    stw  r8,12(r3) // Mem[r3 + 12] <- r8
    bdnz <loopA> // CTR <- CTR - 1, jump to loopA if CTR != 0

<loopB>:
    lwz  r8,12(r3) // r8 <- Mem[r3 + 12]
    mulw r8,r8,r7 // r8 <- r8 * r7
    stw  r8,12(r3) // Mem[r3 + 12] <- r8
    lwz  r3,r3(64) // r3 <- Mem[r3 + 64]
    cmpwi r3,0 // compare r3 with 0
    bne  <loopB> // jump to loopB if r3 != 0

```

Fig. 15. Two loops with different stride properties. In loop A the stride of `r3` is known (64), but is interminable in loop B. In both loops input register `r7` has stride 0.

ses are carried out locally for each basic block and the results are then propagated in global, iterative analysis stages until all values have settled. We then use the obtained register information for each superblock entry in monitoring and recompilation.

Dead Register Utilization. In order for Azure to mark a sequence of basic blocks as potentially optimizable, it needs deterministic information showing that enough registers will be *dead* (i.e. holding values that will never be used before being overwritten) at the point of entry to the leading superblock. The dead registers are utilized in monitoring and as scratch registers for recompiled code. Because of the need to keep overheads very low, we don't want to save and restore registers at superblock boundaries to make room for Azure's required register use, for example in loading thread data to task vectors that are polled by waiting parallel threads. Therefore, Azure does not parallelize regions with fewer than a given number of dead registers available to be used as scratch registers. We have implemented recompilation methods that typically require at most 4 scratch registers, and we have used this number (4) as a threshold. As we have been able to find more than 4 dead registers in almost all superblocks, we have not needed to pay closer attention to this subject, one that would likely have involved implementing a decision mechanism using predicted loop sizes.

Live Register Utilization. All registers found to be *live* at superblock entry and are then used within the superblock are the *input registers* for the superblock. This information is used to identify the parallelizability of regions by analyzing the stride information for input registers. For loop regions involving multiple superblocks, an extra (run-time) analysis stage is required to identify which registers use values defined outside the region, and are therefore input registers for the region.

Stride Information. Azure decides that a loop region is parallelizable if all input registers of the region have either zero strides (read-only) or have fixed strides that are determinable at static analysis-time. This prevents loop regions with input registers that are updated with non-determinable values (such as values loaded from

memory) from being parallelized under Azure. There has been noticeable work done on the locality of load values, but the feasibility studies we have conducted have not shown this to be true in codes considered by Azure for optimization.

Figure 15 shows two code segments, one (loop A) with fixed stride and the other (loop B) with unknown stride. Each code traverses a data structure, multiplying the contents of a data field in each entry by a fixed number. The structure traversed in loop A is an array, whereas the structure traversed in loop B is a linked list.

Both loops have input registers `r3` and `r7`, with the strides calculated by the static analyzer being `{r3:64, r7:0}` for loop A and `{r3:UNKNOWN, r7:0}` for loop B. Azure recognizes the former as parallelizable, the latter as not.

5. RUN-TIME MONITORING AND RECOMPILATION

Our static analyzer extracts a “mostly correct” control-flow graph that is very close to the actual control flow followed by the application at run-time. This “mostly-correct” CFG is then used as the basis for superblock partitioning, and in turn for dynamic instrumentation and recompilation.

Our solution for instrumentation is not entirely dissimilar from more general-purpose mechanisms such as DynInst [Buck and Hollingsworth 2000] or ATOM [Srivastava and Eustace 2004], which also instrument binary executables. However, both our monitoring and recompilation strategies are predicated on the static analyzer having correctly identified single-entry multiple-exit superblock regions.

Hence, our run-time setup forces a switch from a conventional, strictly basic-block based control flow to one that follows the constraints established by the single-entry multiple-exit model. That is, once a control leaves a superblock from one of its exits, the next instruction to be executed should be the entry point of another (or the same) superblock.

However, as has been explained in Section 4, it is possible that there will be control-flow edges that went undetected during the static analysis stage but are triggered during actual execution. Also possible, but less likely, is that some of these edges will target locations in the original binary program that were not declared by the static analyzer to be superblock entries, and therefore *break* the single-entry multiple-exit execution model. As was described above, our static analyzer strives to reduce the probability of the latter occurring by mandating that certain locations be declared as superblock entries even though the control flow graph structure may not strictly demand it. Examples are addresses following `bctrl` instructions, unreachable procedures, and `blr1` instructions, referred to as “must be” entry points in Section 4.5.

Hence, our approach requires that we can correctly and efficiently handle *side entries* to superblocks, i.e. jumps to locations in superblocks other than the entry point. Our static analysis phase reduces the frequency of such events but does not guarantee complete elimination. An efficient run-time handling is required to deal with the few remaining occurrences. Whenever such an exceptional case is detected, the assumptions upon which optimization was based have been invalidated; this needs to be compensated for the current iteration (in which optimized code may already have been executed) and rectified for future iterations (by not again jumping to the wrongly optimized code in the first place).

5.1 Handling Side Entries to Single-Entry Multiple-Exit Regions

Our solution to handling the rare cases of computed branches that invalidate the integrity of a superblock as calculated during static analysis is by keeping the original memory footprint of the input program as a reference point. In particular, we ensure that values of the registers used by indirect branches during execution under Azure are the same as the indirect-branch values during native execution of the original application. On the PowerPC, this applies to the LR and CTR registers.

This is achieved with the use of an *Indirect Branch Table (IBT)* that is mapped to the address space of the original executable, along with modification of branches that set LR to instruction sequences that preserve original values.

The Indirect Branch Table (IBT) is a table that Azure’s loader places in the exact location of the original application code’s virtual address space, interspersed with the code of “released” superblocks. A superblock is “released” either from the very beginning, because it was deemed infeasible for optimization by the static analyzer, or at run-time, because a computed branch was detected that violated the single-entry multiple-exit assumption or it was deemed non-parallelizable. For program regions under the current control of the Azure monitoring and recompilation system, indirect branches will trampoline through the IBT; for all other program regions, an indirect branch will terminate at the original program code.

In essence, one aspect of the IBT (Figure 16) is that of a large hashtable, keyed by (preserved) original indirect branch addresses and storing the new locations of corresponding code fragments. Extending the hashtable analogy, the lookup mechanism is simply an indirect branch to a location in the IBT, which in turn is a direct jump to the new location.

5.2 Code Modifications

Branches set the LR register implicitly by copying the address of the following instruction to the LR as part of a jump instruction. Other instructions that set the LR or CTR registers do not need any change; they would by nature load the registers with values identical to the ones that would be loaded in the original code. Note that CTR registers, when used in indirect branches, can only have been set explicitly. It *is* possible to write malicious code in assembly language that decrements a CTR register until it holds a valid address, but these would be pathological cases that would be inconceivable as “real” code.

Figure 17 shows all instructions that set the LR register, their definitions, and the instruction sequences that replace them in order to preserve the LR values of the original code. For all the entries in the table, *addr_upper* and *addr_lower* represent the sign-adjusted upper and lower 16-bits (respectively) of the original virtual address of the instruction after the branch, register *r0* holds 0, *rX* is a scratch register that was determined to be dead at that location, and the `addis` instruction left-shifts immediate values by 16 bits before addition. Immediate operands in PowerPC instructions can have at most 16 bits.

After these replacements have been inserted, all indirect branches jump not to the modified location of the intended target but to the original target address in the unmodified binary. If the target superblock is “released”, then this is the original

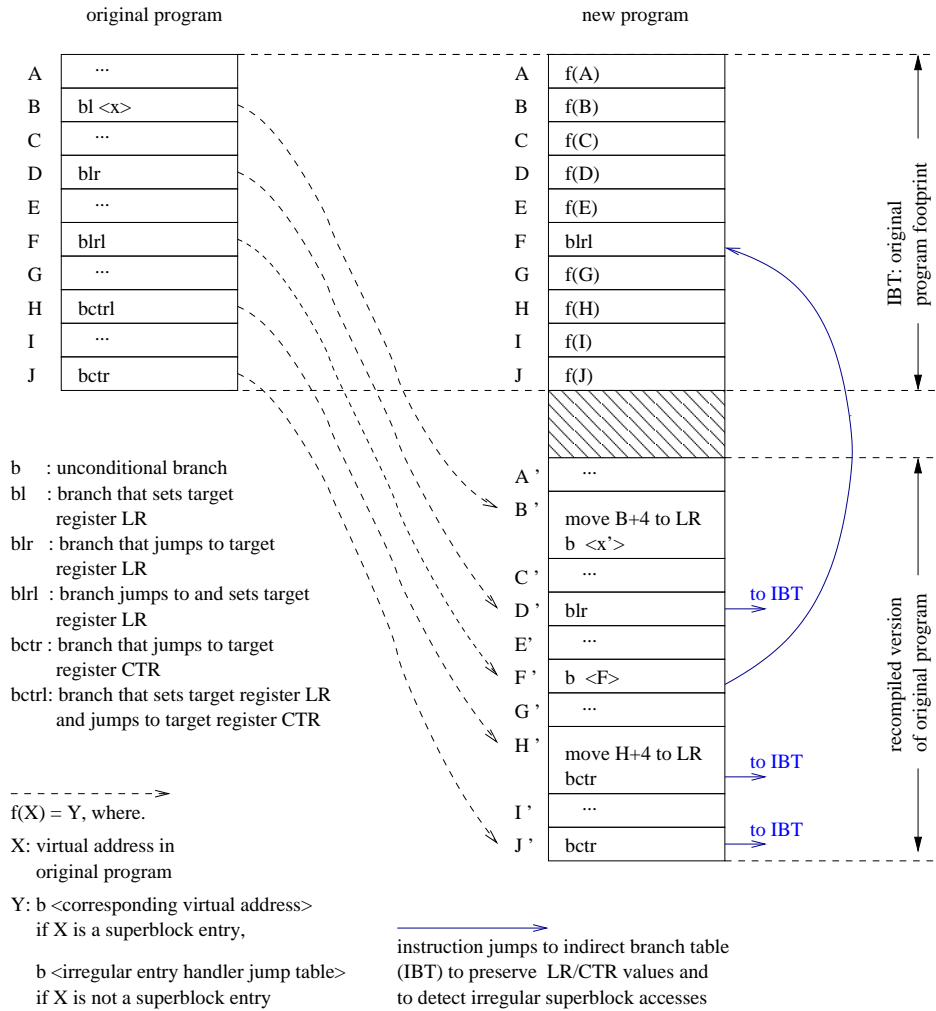


Fig. 16. The Indirect Branch Table.

program code. Otherwise, the destination is part of the Indirect Branch Table.

5.3 Precise and Constant-Time Detection of Side Entries

The Indirect Branch Table provides an extremely efficient way of trapping and handling side entries to superblocks when they occur. A naive, impractical solution would have been to look up all indirect branch targets and check against a list of declared superblocks. Assuming the list would be in a balanced tree structure, the average lookup would be of the order \log_n , with n being the number of superblocks. This is still prohibitively expensive for an operation (an indirect branch) that occurs frequently, but results in the special case (side entry) only very rarely.

Our solution is based on the insight that all indirect branch targets should be superblock entries whose address is known at static-analysis time. In the IBT,

Instruction	Description	New Instruction Sequence
blrl	Jumps to address in LR	addis rX,r0,<addr_upper> addi rX,rX,<addr_lower> mtrl rX blr
bclrl	Jumps to address in LR if the branch condition evaluates to true	bcx [†] +5 [‡] addis rX,r0,<addr_upper> addi rX,rX,<addr_lower> mtrl rX blr
bctrl	Jumps to address in CTR	addis rX,r0,<addr_upper> addi rX,rX,<addr_lower> mtrl rX bctr
bcctrl	Jumps to address in CTR if the branch condition evaluates to true	bcx [†] +5 [‡] addis rX,r0,<addr_upper> addi rX,rX,<addr_lower> mtrl rX bctr
bl <Target>	Jumps to address <i>Target</i>	addis rX,r0,<addr_upper> addi rX,rX,<addr_lower> mtrl rX b <Target>
bcl <Target>	Jumps to address <i>Target</i> if the branch condition evaluates to true	bcx [†] +5 [‡] addis rX,r0,<addr_upper> addi rX,rX,<addr_lower> mtrl rX b <Target>

Fig. 17. LR-setting Instructions in the original binary and their replacement sequences that preserve original LR values. *addr_upper* and *addr_lower* are the upper and lower 16 bits (adjusted for signed addition) of the original address of the next instruction. *rX* is a scratch register, *r0* is zero, and the `addis` instruction is a shift-add-immediate with 16-bit shifts. [†]Conditional branch with the inverted condition code. [‡]If taken, this instruction jumps to the 5th instruction after itself.

we place an *irregular access handler* into all addresses that don't correspond to superblock headers. We thereby remove the need for trapping superblock exits and querying target addresses, and we can detect side entries at constant time without incurring any additional overhead for valid indirect branch accesses. This also allows us to overlay “released” code into the address range of the IBT itself (which is where it originated in the original program).

Once an irregular entry occurs, control is handed over from the IBT to the Azure system's side entry handler. This is implemented in a slightly more complicated way than depicted in Figure 16, which presents a simplified view of the IBT mechanism. Basically, locations in the IBT that are not superblock entries hold branch instructions that jump to a table for the side entry handler which first marks the location of the side entry, and then jumps to the actual side entry handler. This ensures that all side entries are caught at the correct instruction, whether the instruction is at the start or middle of a basic block.

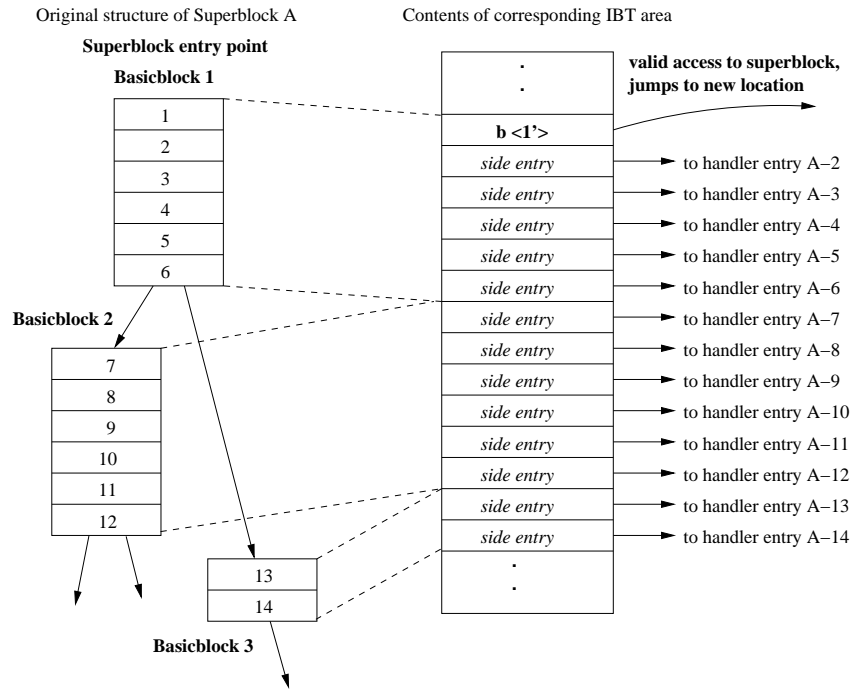


Fig. 18. Locations of access to the side entry handler. Accesses are made via a jump table that ensures the side entry handler knows the exact location of the side entry to ensure correct execution at all times.

The side entry handler mechanism of Azure maintains correctness in the presence of side entries on a single-entry multiple-exit-based execution model by removing the superblocks that are side entry targets from the confines of the single-entry multiple-exit model. This is accomplished by overwriting the contents of the original superblocks to the corresponding locations in the IBT table. All subsequent accesses to the same superblock (side entry or not) therefore result in execution of the original superblock code in the location previously used as the IBT table (which in turn used the virtual addresses of the superblock in the original application). The superblock is then released from future attention of the run-time system.

In order to ensure correctness, the system has to know exactly at which location the side entry occurred, so as to allow continuation of the application's execution at the correct point, without re-executing or skipping any instructions. This is maintained by another jump table structured as depicted in Figure 18.

Figure 19 shows a typical code sequence in the jump table between the IBT and the side entry handler that resides in the Azure code space. Register \mathbf{rX} is a scratch register for the superblock. It should be noted that in this context, dead registers are not the same as scratch registers; any instruction can be a side entrance and a register dead at superblock entry may be live after some superblock instruction. Therefore a scratch register has to be dead at superblock entry and unused within the superblock.

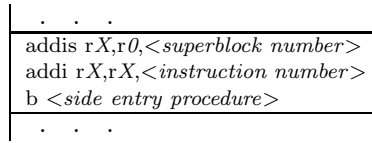


Fig. 19. Typical code sequence in the jump table between the IBT and the side entry handler in the Azure code space. Register `rX` is a scratch register of the superblock and register `r0` is zero in this kind of usage.

In principle, we need to have side entry jump-table entries for each instruction that is not a superblock entry. Without any modifications, this would result in extra memory usage of around slightly less than three times the code in question, since each entry holds two instructions. One instruction overwrites the superblock number to the upper 16-bit of a scratch register of the superblock, and a second instruction that writes the instruction number of the side entry instruction in the superblock to the lower 16-bit. The instruction number is in fact the signed address offset of the instruction from the entry address, right shifted by two bits.

The rare cases of applications with more than 64K superblocks (none of our application benchmarks were seen to have more than 16K superblocks after static analysis), and those having superblocks with instructions at a distance of more than 128Kbytes away, are handled by overwriting two scratch registers instead of one. If not enough scratch registers are available, the superblock is released.

As side entries occur highly infrequently (often never), overwriting the corresponding IBT addresses, invalidating the corresponding I-cache and ensuring that execution continues from the correct instruction does not cause a noticeable performance degradation. Even when they do occur, side entries tend to be caused by a small number of instruction-target pairs and only a few calls are made to the side entry handler (the corresponding superblocks are then released). Executed with the longest-running inputs of all our benchmarks, the number of calls to the Azure side entry handler never exceeded two-digit figures.

Space Overhead. 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. A benefit of having the indirect branch table fill the entire original code range is that once a decision has been made (statically or dynamically) that a certain code region is not feasible for parallelization, the initial address range of the region inside the IBT can simply be overwritten by the original code.

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. As shown in Figure 13, the number of feasible superblocks remains less than 500 in all benchmarks after the static elimination stage. 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 can be measured in the tens of kilobytes for all cases.

There are other runtime structures as well, required for the implementation of various application-to-Azure control transfers (handling side entries, profiling, in-

sertion of recompiled code), but these do not contribute significantly to the overall memory footprint. The IBT and Azure system code together increase the load-time memory utilization by a factor of at least four, depending on the size of the binary executable. The sizes of the IBT and the recompiled code region are proportional to the input executable's size, but the Azure system code size is fixed (slightly less than 1MB in the current version).

5.4 Identification and Optimization of Parallelizable Regions

A detailed explanation of our experiences with various dynamic methods of identifying parallelizable regions is beyond the scope of this paper, as are the techniques we employ for dynamic recompilation and thread management. We refer the readers to the first author's doctoral dissertation [Yardımcı 2006] for more information.

The overall system behavior is as follows: The Azure runtime uses profiling instructions inserted at the entrance of each candidate superblock to identify parallelizable loop regions. These regions could be composed of a single superblock or loops formed by more than one superblock. Once a loop has been identified as a parallelizable region (and has passed a parallelization legality check), it is monitored for several executions of a loop. If the average iteration count exceeds the parallelization threshold (described in Section 3.3), parallelization of the loop is considered feasible and the Azure runtime replaces the profiling code with a jump to a parallelized version of the code emitted by the recompilation engine. We speculate that a region seen to have a high iteration count average will display similar characteristics in subsequent accesses during the execution.

We would like to reemphasize at this point that the static loop elimination method we employ guarantees that there will be no data dependence for a given region, as we do not consider regions for which we cannot deterministically state the impossibility of data dependence violations. The regions that have been eliminated during the static phase will have been stripped of profiling hooks. Figure 13 displays the number of remaining eligible superblocks. Although the number of eligible superblocks is low relative to the total number, the dynamic coverage exceeds 30% in most cases.

In order to be able to exploit parallelism found in recursive loop iterations, we also 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.

6. EVALUATION AND RESULTS

Azure can currently be deployed on 32-bit PowerPC (PPC32) and 64-bit POWER (PPC64) platforms using the GNU/Linux operating system with a kernel greater than 2.6. Linux kernels 2.6 and above provide the facility to tie a given process to a specific processing element through the `sched_set_affinity` system call; this is a critical capability required by our system to provide mapping of parallel threads to available processing element resources. The two versions of Azure for PPC32

Benchmark	Suite	Program Language	Description
gzip	SPEC CPU2000 Integer	C	Compression
vpr	SPEC CPU2000 Integer	C	Circuit Placement and Routing
mcf	SPEC CPU2000 Integer	C	Combinatorial Optimization
parser	SPEC CPU2000 Integer	C	Word Processing
bzip2	SPEC CPU2000 Integer	C	Compression
twolf	SPEC CPU2000 Integer	C	Place and Route Simulator
perlbnk	SPEC CPU2000 Integer	C	PERL Programming Language
art	SPEC CPU2000 FP	C	Image Recognition/Neural Networks
swim	SPEC CPU2000 FP	Fortran	Shallow Water Modeling
treeadd	Olden	C	Recursive sum in balanced B-tree
em3d	Olden	C	Electromagnetic wave propagation

Fig. 20. Description of the benchmarks used in our experiments.

and PPC64 are almost identical and differ mainly in memory addressing.

We benchmarked our system on a dual 866MHz G4 processor Apple Power Macintosh, running a Linux OS with a 2.6.8 kernel, and also on a 1.67 GHz POWER5 system with SMT and CMP capabilities. This enabled us to extrapolate how the system will perform on a wide range of parallel hardware, with different proximities between processing elements. No simulation was used. The benchmarks were chosen from a mix of SPEC CPU2000 Integer (gzip to parser), SPEC CPU2000 floating-point (swim and art) and Olden benchmarks (treeadd and em3d). We were not able to run all the benchmarks we used in our static analyses due to initialization problems inside the loader routine. However, we are satisfied with the resulting selection set (presented in Figure 20), as the applications in it form a representative and diverse mix. Although there is an imbalance against floating-point benchmarks in this mix, we would like to note that Integer benchmarks are in fact more challenging due to their more complex control flows. This highlights one of the key aims of the Azure system, which is using dynamic control-flow information to parallelize applications that static compilers find hard to optimize.

All benchmarks were compiled with GCC 3.2.2 with full optimization and executed with each benchmarks' reference inputs and command line arguments.

Figure 21 shows the results obtained on the dual PowerPC processor. The striped bars show 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. 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 display the best results, having simpler structures with parallelizable loops accounting for most of the execution time. 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 (using

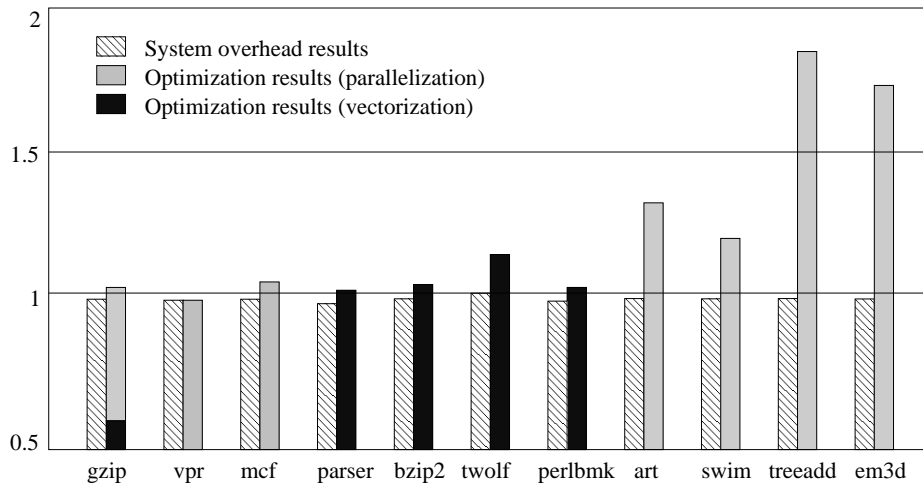


Fig. 21. Effects of our system on the performance of statically optimized binary programs, on a dual-processor PowerPC platform. All optimizations break even except for vpr, for which no optimizable region was found. The overall result for vpr corresponds to the overhead of the Azure system.

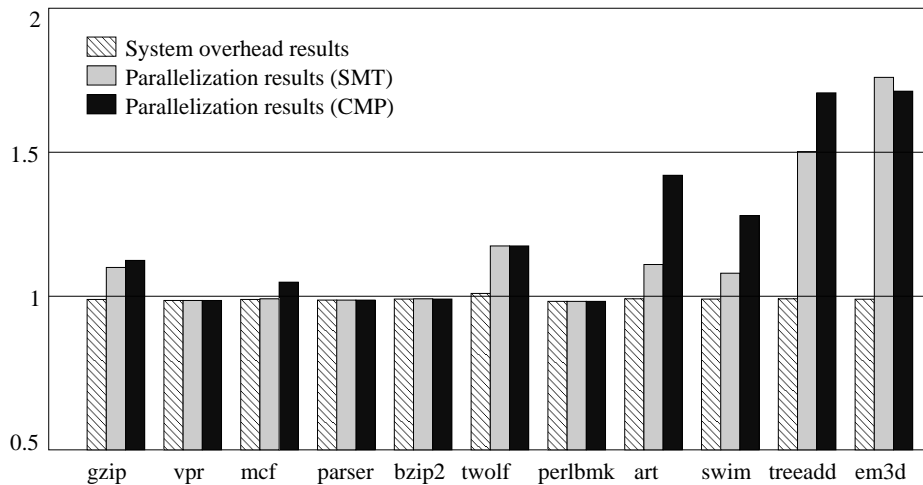


Fig. 22. The same benchmarks on two active processors on a 4x4 POWER5. SMT denotes results in which two logical processors reside on the same core, CMP in which the two processors reside on different cores communicating through the L2 cache.

dynamically observed data widths). Within the Integer benchmarks, most of the improvement results were obtained through vectorization. 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 fewer than 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 Azure was able to improve on the PowerPC platform register a slight slowdown (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 almost twice the amount of physical hardware resources 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 than 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-mapping on the POWER5; this can be explained by the large loop size (16M iterations) due to the reference arguments for the treeadd benchmark.

6.1 Run-Time System Overheads

Despite the overheads of profiling, of the extra instructions needed to preserve indirect branch target register values, the extra jump instructions incurred by the usage of the indirect branch table, and utilization of far more virtual memory address space than was requested by the original binary, the overall performance degradation caused by executing alongside the Azure system is very small (please refer again to Figure 14).

This remarkable performance is primarily due to the off-line static analysis phase, in which the program is partitioned imprecisely into single-entry multiple-exit superblock regions ahead of time. Superblocks that are not feasible for optimization (for various reasons) are recognized at this time and are never again touched by the dynamic part of the system (in terms used elsewhere in this paper, they are “released”).

Contributing factors to overheads are the role carried out by the virtual memory system and the tendency of programs to spend most of the execution times running a small percentage of all code (a situation generally referred to as the 90/10 rule, that is, 90% of a program’s execution taking place in 10% of the program’s code). The disproportionate execution time of small parts of the code implies that most of the virtual address space mapped to hold the indirect branch table, the side entry jump table, and other structures required to support execution mostly do not reside in the cache, and are possibly paged out by the memory system. Hence, cache-aware tuning of these structures might reduce overheads even further.

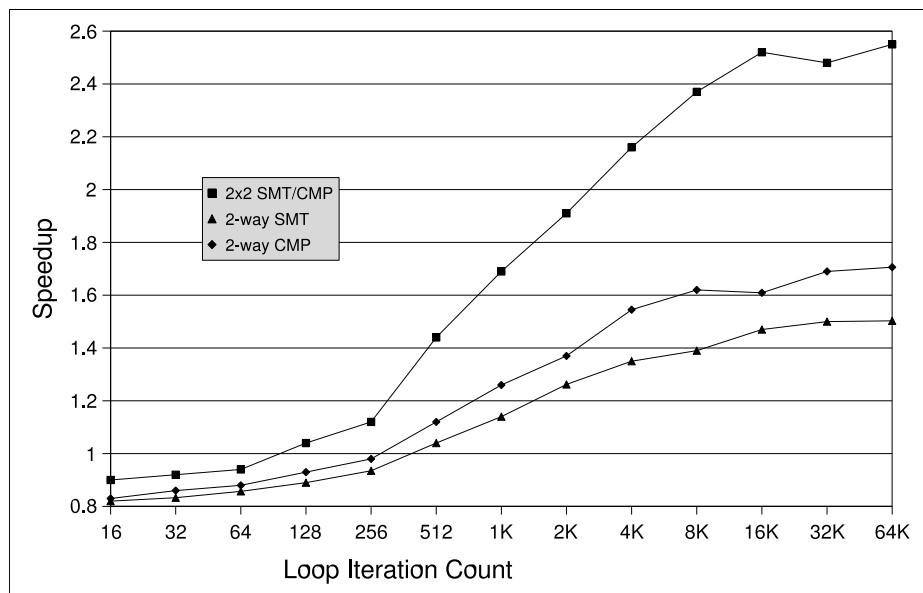


Fig. 23. `em3d` speedups for various loops sizes on 2-way CMP, 2-way SMT and 4-way (2×2) SMT/CMP platforms

6.2 Scalability Studies

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

In general, two factors inhibit scalability of Azure. Mentioned before in Section 3.3, a important 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 original iteration count required to surpass the synchronization overhead is doubled, even if the overheads are otherwise equal. Furthermore, the communication overhead between parallel threads is also increased.

The performance results displayed in Figures 23 and 24 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 2×2 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 23 that a speedup plateau is reached when the iteration counts exceed a certain limit, beyond 8K in the case of `treeadd`. Speedup results decrease as the iteration averages are reduced, falling below 2 (on a 4-way system) when it is reduced to 2K. For averages of less than 512 iterations, parallelizing `treeadd` on 4 threads achieves results not significantly better than 2-way parallelizations.

Similar results have been obtained for `em3d`. Speedups of between 2.2 and 2.5 are obtained for when the iteration count is equal to or above 400. Speedups dramatically decrease when the iteration count is decreased to 200, to the same level

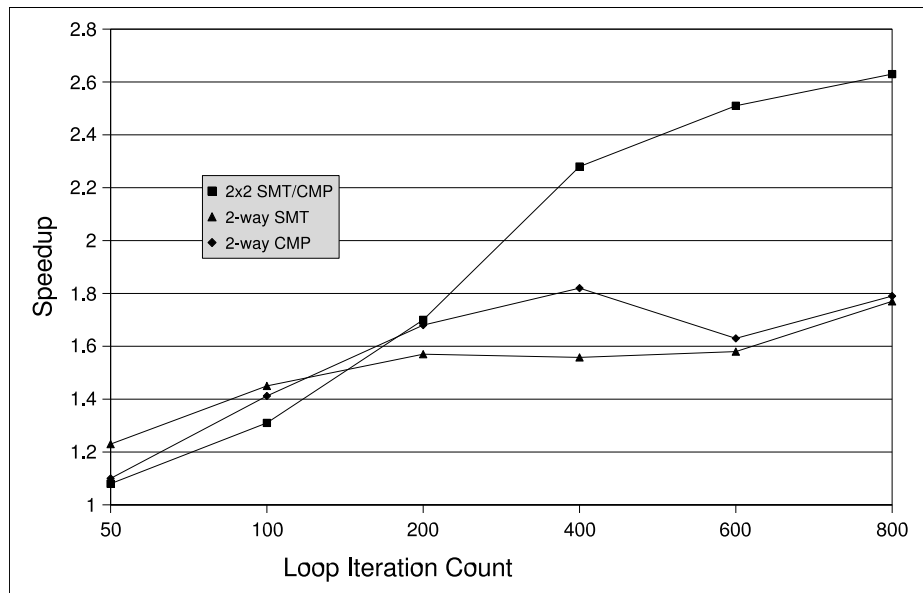


Fig. 24. `treeadd` speedups for various loops sizes on 2-way CMP, 2-way SMT and 4-way (2x2) SMT/CMP platforms

as 2-way parallelization, and *fall below* the 2-way results with further decreases.

We do not expect to see Azure providing a comprehensive mechanism to enable parallelization on all types of applications. Certain applications with obviously parallel code will not require run-time monitoring, and certain applications with complex inter-thread communication requirements will 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.

As a result, we never expected for Azure to be scalable beyond reasonable, modest limits. Our expectations, which are met by experimental results, were to achieve significant performance improvements on platforms having up to 8, but typically between 2 or 4 processing elements.

7. CONCLUSIONS AND OUTLOOK

Azure is an unintrusive, entirely software-based system that utilizes cheaply available parallel resources to extract performance improvements from the residual parallelism in uniprocessor applications. It does this for precompiled applications with no symbol table or source code information, and is even agnostic with respect to the application binary interface.

Key to the remarkably low overhead is a single-entry multiple-exit model of execution regions. The underlying analysis is performed ahead of time in an off-line step. The results of this analysis, while good, are possibly incorrect in a way that might undermine the single-entry multiple-exit execution paradigm. However, we are able to detect these cases 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, Azure is able to usually “break even”, or even deliver significant performance improvements to most benchmarking workloads. Perhaps just as importantly, 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 obtained so far, we aim to continue this research by investigating how the system would scale. We are especially optimistic on what can be achieved with floating point and media applications. Other future work includes incorporation of data speculation into our decision structure, and adapting our system to specialized hardware solutions such as the Stanford Hydra Microarchitecture [Hammond et al. 2000]. We hope to eventually be able to parallelize a much greater percentage of loops than we can now.

We see no reason why an Azure-like system cannot be implemented on an architecture other than PowerPC. Porting Azure to another RISC-like architecture would be straightforward, while porting to a CISC ISA such as x86 would be more difficult but still feasible. An x86 implementation would require more complicated methods of offset adjustment and recompilation to handle instructions with non-uniform widths. The reduction in available general-purpose registers (down from 32 to 16 in x86-64) would be problematic, but this would be compensated by the expanded set of available instructions. Another benefit would be being able to load scalar values directly to SSE registers without having to go through the memory interface (values can be transferred to PowerPC AltiVec registers only through vector load instructions).

We expect that there will be a place for systems such as Azure even when in the medium term software developers will focus their attention on explicit parallelism. First, it will enable a more gradual transition that does not require all existing programs to be rewritten radically to profit from parallel features in the hardware. And second, our approach requires only one version of a binary for good execution performance across a wide range of hardware, including uniprocessors. This also implies a new lease of performance for older legacy software.

REFERENCES

- AKKARY, H. AND DRISCOLL, M. A. 1998. A dynamic multithreading processor. In *Proceedings of the 31st Annual International Symposium on Microarchitecture*. ACM Press, Dallas, TX, USA, 226–236.
- BALA, V., DUESTERWALD, E., AND BANERJIA, S. 2000. Dynamo: A transparent dynamic optimization system. In *Proc. of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*. 1–12.
- BALAKRISHNAN, G. AND REPS, T. 2004. Analyzing memory accesses in x86 executables. In *Proc. Compiler Construction (LNCS 2985)*. Springer Verlag, 5–23.
- BARAZ, L., DEVOR, T., ETZION, O., GOLDENBERG, S., SKALETSKY, A., WANG, Y., AND ZEMACH, Y. 2003. IA-32 Execution Layer: a two-phase dynamic translator designed to support IA-32. *ACM Transactions on Programming Languages and Systems*, Vol. V, No. N, April 2008.

- applications on Itanium-based systems. In *Proceedings of the 36th International Symposium on Microarchitecture*. IEEE, 191–201.
- BUCK, B. AND HOLLINGSWORTH, J. K. 2000. An API for runtime code patching. *The International Journal of High Performance Computing Applications* 14, 4 (Winter), 317–329.
- BYRD, G. T. AND HOLLIDAY, M. A. 1995. Multithreaded processor architecture. *IEEE Spectrum* 32, 8 (Aug.), 38–46.
- CARLISLE, M. C., ROGERS, A., REPPY, J. H., AND HENDREN, L. J. 1994. Early experiences with olden. In *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*. Springer-Verlag, London, UK, 1–20.
- CHAMBERS, C. 2002. Staged compilation. In *PEPM '02: Proceedings of the 2002 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation*. ACM Press, New York, NY, USA, 1–8.
- CHERNOFF, A., HERDEG, M., HOOKWAY, R., REEVE, C., RUBIN, N., TYE, T., YADAVALLI, S. B., AND YATES, J. 1998. FX!32: A profile-directed binary translator. *IEEE Micro* 18, 2 (Mar/Apr), 56–64.
- CIFUENTES, C. AND GOUGH, K. J. 1995. Decompilation of binary programs. *Softw. Pract. Exper.* 25, 7, 811–829.
- CINTRA, M. AND LLANOS, D. R. 2003. Toward efficient and robust software speculative parallelization on multiprocessors. In *PPoPP '03: Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM Press, New York, NY, USA, 13–24.
- DE ALBA, M. AND KAEI, D. 2001. Runtime predictability of loops. In *4th Annual IEEE International Workshop on Workload Characterization*.
- EBCIOĞLU, K. AND ALTMAN, E. R. 1997. DAISY: Dynamic compilation for 100% architectural compatibility. In *24th Annual International Symposium on Computer Architecture*. 26–37.
- EBCIOĞLU, K., FRITTS, J., KOSONOCKY, S., GSCHWIND, M., ALTMAN, E., KAILAS, K., AND BRIGHT, T. 1998. An eight-issue tree VLIW processor for dynamic binary translation. In *Proceedings of ICCD-98*.
- FISHER, J. A. 1981. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers* 30, 7 (July), 478–490.
- GRANT, B., PHILIPPOSE, M., MOCK, M., CHAMBERS, C., AND EGGERS, S. J. 1999. An evaluation of staged run-time optimizations in DyC. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*. ACM Press, 293–304.
- HAMMOND, L., HUBBERT, B. A., SIU, M., PRABHU, M. K., CHEN, M., AND OLUKOTUN, K. 2000. The Stanford Hydra CMP. *IEEE Micro* 20, 2, 71–84.
- HWU, W., MAHLKE, S., CHEN, W., CHANG, P., WARTER, N., BRINGMANN, R., OUELLETTE, R., HANK, R., KIYOHARA, T., HAAB, G., HOLM, J., AND LAVERY, D. 1993. The superbloc: an effective technique for vliw and superscalar compilation. *The Journal of Supercomputing* 7, 1–2, 229–248.
- KAGAN, M., GOCHMAN, S., ORENSTIEN, D., AND LIN, D. 1997. MMX microarchitecture of Pentium processors with MMX technology and Pentium II microprocessors. *Intel Technology Journal*, 8.
- KISTLER, T. AND FRANZ, M. 2001. Continuous program optimization: Design and evaluation. *IEEE Transactions on Computers* 50, 6 (June), 549–566.
- KISTLER, T. AND FRANZ, M. 2003. Continuous program optimization: A case study. *ACM Transactions on Programming Languages and Systems* 25, 4, 500–548.
- KLAIBER, A. 2000. The technology behind Crusoe processors. In *Transmeta Technical Brief*.
- KRISHNAN, V. AND TORRELLAS, J. 1998. Hardware and software support for speculative execution of sequential binaries on a chip-multiprocessor. In *International Conference on Supercomputing*. 85–92.
- KRISHNAN, V. AND TORRELLAS, J. 1999. A chip-multiprocessor architecture with speculative multithreading. *IEEE Transactions on Computers* 48, 9, 866–880.
- KRISHNAN, V. S. 1998. Speculative multithreading architectures. Tech. Rep. UIUCDCS-R-98-2048, UIUC.

- LARUS, J. R. AND BALL, T. 1994. Rewriting executable files to measure program behavior. *Softw. Pract. Exper.* 24, 2, 197–218.
- LEUNG, A. AND GEORGE, L. 1999. Static single assignment form for machine code. In *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*. ACM Press, 204–214.
- LO, J. 1998. Exploiting thread-level parallelism on simultaneous multithreaded processors. Ph.D. thesis, University of Washington.
- NGUYEN, H. AND JOHN, L. K. 1999. Exploiting SIMD parallelism in DSP and multimedia algorithms using the altivec technology. In *International Conference on Supercomputing*. 11–20.
- OBERMAN, S., FAVOR, G., AND WEBER, F. 1999. AMD 3DNow! technology: Architecture and implementations. *IEEE Micro* 19, 2 (Mar./Apr.), 37–48.
- OLUKOTUN, K., NAYFEH, B. A., HAMMOND, L., WILSON, K., AND CHANG, K. 1996. The case for a single-chip multiprocessor. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*. 2–11.
- QUINONES, C. G., MADRILES, C., SANCHEZ, J., MARCUELLO, P., GONZALEZ, A., AND TULLSEN, D. M. 2005. Mitosis compiler: an infrastructure for speculative threading based on pre-computation slices. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, New York, NY, USA, 269–279.
- RAUCHWERGER, L. AND PADUA, D. A. 1999. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. *IEEE Trans. Parallel Distrib. Syst.* 10, 2, 160–180.
- ROTENBERG, E., JACOBSON, Q., SAZEIDES, Y., AND SMITH, J. 1997. Trace processors. In *International Symposium on Microarchitecture*. 138–148.
- SKOGLUND, J. AND FELSBURG, M. 2005. Fast image processing using sse2. In *Proceedings of the SSBA Symposium on Image Analysis*. Malmö.
- SOHI, G. S., BREACH, S. E., AND VIJAYKUMAR, T. N. 1998. Multiscalar processors. In *25 Years of ISCA: Retrospectives and Reprints*. 521–532.
- SRIVASTAVA, A. AND EUSTACE, A. 2004. Atom: a system for building customized program analysis tools. *SIGPLAN Not.* 39, 4, 528–539.
- THAKKAR, S. T. AND HUFF, T. 1999. The Internet Streaming SIMD Extensions. *Intel Technology Journal*, 8.
- TSAI, J.-Y., HUANG, J., AMLO, C., LILJA, D. J., AND YEW, P.-C. 1999. The superthreaded processor architecture. *IEEE Transactions on Computers* 48, 9, 881–902.
- TSAI, J.-Y. AND YEW, P.-C. 1996. The superthreaded architecture: Thread pipelining with run-time data dependence checking and control speculation. In *Parallel Architectures and Compilation Techniques*. 35–46.
- TULLSEN, D. M., EGGERS, S., AND LEVY, H. M. 1995. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22th Annual International Symposium on Computer Architecture*. 392–403.
- VOSS, M. J. AND EIGENMANN, R. 2000. Adapt: Automated de-coupled adaptive program transformation. In *ICPP '00: Proceedings of the Proceedings of the 2000 International Conference on Parallel Processing*. IEEE Computer Society, Washington, DC, USA, 163.
- VOSS, M. J. AND EIGENMANN, R. 2001. High-level adaptive program optimization with ADAPT. *ACM SIGPLAN Notices* 36, 7 (July), 93–102.
- YARDIMCI, E. 2006. Exploiting parallelism to improve the performance of sequential binary executables. Ph.D. thesis, University of California, Irvine.
- YARDIMCI, E. AND FRANZ, M. 2006. Dynamic parallelization and mapping of binary executables on hierarchical platforms. In *CF '06: Proceedings of the 3rd conference on Computing frontiers*. ACM Press, New York, NY, USA, 127–138.
- ZILLES, C. AND SOHI, G. 2001. A Programmable Co-processor for Profiling. In *Proc. 7th International Symposium on High Performance Computer Architecture*.