Execution Migration in a Heterogeneous-ISA Chip Multiprocessor

Matthew DeVuyst   Ashish Venkat   Dean M. Tullsen
University of California, San Diego
{mdevuyst | asvenkat | tullsen}@cs.ucsd.edu

Abstract

Prior research has shown that single-ISA heterogeneous chip multiprocessors have the potential for greater performance and energy efficiency than homogeneous CMPs. However, restricting the cores to a single ISA removes an important opportunity for greater heterogeneity. To take full advantage of a heterogeneous-ISA CMP, however, we must be able to migrate execution among heterogeneous cores in order to adapt to program phase changes and changing external conditions (e.g., system power state).

This paper explores migration on heterogeneous-ISA CMPs. This is non-trivial because program state is kept in an architecture-specific form; therefore, state transformation is necessary for migration. To keep migration cost low, the amount of state that requires transformation must be minimized. This work identifies large portions of program state whose form is not critical for performance; the compiler is modified to produce programs that keep most of their state in an architecture-neutral form so that only a small number of data items must be repositioned and no pointers need to be changed. The result is low migration cost with minimal sacrifice of non-migration performance.

Additionally, this work leverages binary translation to enable instantaneous migration. When migration is requested, the program is immediately migrated to a different core where binary translation runs for a short time until a function call is reached, at which point program state is transformed and execution continues natively on the new core.

This system can tolerate migrations as often as every 100 ms and still retain 95% of the performance of a system that does not do, or support, migration.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors – Compilers; Code Generation

General Terms Design, Performance, Languages

Keywords Heterogeneous CMP, Thread migration

1. Introduction

Modern general-purpose processors contain multiple cores on a single die, and industry roadmaps call for increasing core counts. Current industry offerings are homogeneous CMPs (all cores on a die are identical); however, research has shown that single-ISA (instruction set architecture) heterogeneous CMPs can achieve even greater performance and power efficiency [12, 16]. Restricting the cores to a single ISA, however, eliminates an important dimension of potential heterogeneity. ISAs are already designed to meet different goals: some are designed to make hardware implementation simple, to reduce code size, to reduce memory accesses, to enable more energy-efficient hardware design, or to support domain-specific instructions. Heterogeneous-ISA CMPs would allow architects flexibility to create more efficient multicore microprocessors for mixed workloads. Additionally, support for multi-ISA execution enables the design and use of ISAs that are less general-purpose, targeted at a particular application domain or desired execution characteristics.

There are several obstacles that must be overcome to make general-purpose heterogeneous-ISA CMPs feasible. This paper proposes a solution to the most significant barrier: process migration. The ability to migrate a running program among heterogeneous cores is critical because it allows the system to capitalize on the available heterogeneity by being able to adapt to both phase changes and environmental changes. Reasons that a thread may want to migrate include:

- When the power state of the computer changes (e.g., a laptop going from normal to low-battery operation), programs running on a core designed for high performance could be migrated to a core designed for energy efficiency.
- When a new process with high priority and high performance demands enters the run queue, other processes could be migrated away from the most powerful core.
- If a program enters a new phase of execution with different computational demands (e.g., floating-point intensive code or cryptographic code), execution could migrate to a core with strong support for the new computation type (e.g., dedicated vector hardware or cryptographic instructions).
- If a part of the chip becomes too hot, programs executing on cores in that region could migrate away. If the cooler cores are a different ISA, threads can still make progress.

Many special-purpose heterogeneous multiprocessors have been produced [5, 9, 27]. Called heterogeneous multiprocessor systems-on-chips (MPSoCs), these architectures are designed for computation in specific domains that require a diverse set of algorithms to solve a problem. This paper focuses on migration in the context of general-purpose heterogeneous CMPs. Migration in this context is more important than on embedded systems where hardware-software co-design is more common and flexibility is less important. Nevertheless, migration can benefit heterogeneous MPSoCs as well by easing programmer burden.

Execution migration on a heterogeneous-ISA CMP is much more challenging than on a homogeneous CMP because program state (in registers and memory) is kept in an architecture-specific

ASPLOS'12, March 3–7, 2012, London, England, UK.
Copyright © 2012 ACM 978-1-4503-0759-8/12/03. . . $10.00

form. This requires state transformation during migration, which can be very expensive.

The goal of this research is dramatic reduction in the cost of state transformation at migration time. Prior work on migration among heterogeneous systems has only considered migration among heterogeneous machines (e.g., among nodes in a grid), not among heterogeneous cores in a CMP. In all of those cases, migration cost was dominated by state copy between nodes; however, on a multicore, that copy is unnecessary because cores share memory, opening the door for fast and frequent migration. But this now exposes the cost of the state transformation step. Because this cost was a small factor in cross-machine migration, the optimizations and transformations studied in this work are largely unique to our approach.

Thus, to minimize migration time, it is essential that as little transformation as possible be done. We accomplish this by ensuring that the memory state of the program, at particular points throughout execution, is nearly identical across compilations for different architectures. During migration, most data objects do not need to be copied or repositioned. As a result, pointers remain correct after migration, avoiding the cost and complexity of finding and fixing pointers at migration time.

We cannot maintain points of equivalence (where memory state is consistent between the executables) at all instructions in the program without serious performance consequences (e.g., worse than unoptimized code). We relax the equivalence constraint to only those points in execution that represent function call sites. However, we still allow for instantaneous migration (i.e., at any instruction) by performing binary translation on the migrated process until a function call is reached, at which point the program state is quickly transformed and execution is resumed natively. Thus, by adding the ability to do binary translation, we decouple the two aspects of migration—we can restrict migration between ISAs to specific equivalence points (call sites), yet still support migration between cores at any arbitrary point in the program.

In addition to the challenges presented by ISA diversity, the fact that migration, and hence binary translation, can start at random points of execution and run for a short duration (until a function call is reached) poses a challenge to efficient binary translation. We discuss these challenges and present an efficient dynamic binary translation and optimization technique in this paper.

If we can limit the incremental performance cost for cross-ISA migration to a few percent, even assuming very frequent migration (e.g., every few hundred milliseconds), then there is no significant performance barrier to using multi-ISA heterogeneous cores, making the full spectrum of heterogeneity available to processor designers. Besides minimizing performance cost (both for migrating code as well as steady-state non-migrating code), we have the following additional goals for this design. The technique should not require type-safe code, should not require special effort by the programmer, and should not require any special hardware. Migration should be possible at any instruction.

The level of ISA heterogeneity between cores could vary widely, from ISA extensions that are specific to a subset of the cores, to completely different ISAs. To maximize the generality of our system, we tackle the hardest problem, of completely disjoint ISAs. However, most of the techniques and principles of this work would also apply when the degree of heterogeneity is more moderate.

The next section discusses related work in the area of execution migration as it has been applied to other heterogeneous domains. In Section 3 we give an overview of the migration strategy. Section 4 describes how we ensure a nearly identical memory image across compilation for different architectures to make migration fast. Section 5 describes migration itself. Section 6 describes the binary translation techniques used to allow for instantaneous migration. Section 7 details our experimental methodology, while Section 8 shows the performance of migration. Section 9 concludes.

2. Related Work

Kumar, et al. [15, 17] demonstrate that a CMP of heterogeneous cores (with a homogeneous ISA) is often the best use of die area for better performance and power efficiency on mixed workloads. Removing the restriction of a single ISA among all cores allows for greater heterogeneity and the potential for more efficiency. In the embedded world heterogeneous-ISA multiprocessors (MPSoCs) are not uncommon [5, 9, 27], but no truly general-purpose heterogeneous-ISA CMPs are on the market at this time. The Cell processor [13] lies somewhere in the middle—more general-purpose than most embedded systems, but not designed for mainstream mixed workloads. It has no ability to migrate code between core types dynamically.

To our knowledge, no previous work has addressed the problem of migrating a process among heterogeneous-ISA cores on a chip multiprocessor. However, there are two closely related problems, both potentially involving machines of different architectures: process migration on a cluster, grid, or distributed system; and checkpointing and recovery (CPR). The related works described in this section address one or both of these problems. In every proposed solution, large-scale state copy is necessary and dominates latency. Thus, it is typically the primary performance concern.

Von Bank, et al. [29] put forth a theoretical model, including a formal definition of heterogeneous process migration. They assert that the language system (i.e., compiler, assembler, linker, etc.) can be designed to ensure points of equivalence at a desired granularity; but the finer the granularity, the more performance will suffer because the compiler will be more constrained and less able to make machine-dependent optimizations—a conclusion that we confirm empirically in this research.

Dubach and Shub [8, 21] produce some of the earliest work on process migration. Their solution is meant to work on the equivalent of a strongly-typed language to facilitate low-cost state transformation. They follow a principle they call "greatest common denominator" (GCD), where data is placed in memory with extra padding, if necessary, to accommodate the largest data representation among architectures to which execution might be migrated. This allows most pointers to work without the need for correction. We apply the same GCD principle to our work; only in our case, the migrated process uses the exact same memory image instead of a copy of it. Also, our compiler ensures that function entry points appear at the same offsets within the code sections so that function pointers do not need to be located and fixed.

Several researchers have proposed migration techniques designed to work with strongly-typed programming languages, like Java [28] and Emerald [24]. Many more have proposed migration and/or CPR through the instrumentation of well-typed C code [4, 10, 11, 14, 19, 23, 25, 26]. The definition of well-typed code is different among these works, with some work supporting certain type-unsafe constructs. In contrast, the migration technique this paper proposes does not place any such restrictions on the code.

Pointers in languages like C present a difficult problem for heterogeneous migration because they are machine-dependent, allowing direct access to any object in the address space of a process. No prior work has been able to efficiently deal with the problem of pointers. Either runtime performance [14] or migration performance [10, 11, 19, 23, 25, 26] is sacrificed.

One of the main contributions of our work is to demonstrate an effective method of dealing with pointers (by ensuring all pointed-to objects remain at fixed locations).

Traditionally, binary translation has been used to convert legacy binaries from one ISA to another. Chen, et al. [6] describe a translation technique for translating ARM binaries for execution on a MIPS-like architecture. This work deals with similar challenges of ISA diversity. However, their work describes static translation while our work proposes dynamic translation, requiring different approaches in many cases.

Some examples of dynamic translators used for emulation include Digital's FX!32 [20] (which translates x86 applications to Alpha) and HP's Aries [30] (which translates PA-RISC applications to IA-64). These translators use a two-phase translation, where the first phase does emulation and collects runtime profile information, and the second performs optimization. We cannot afford to have a two-phase translation, as binary translation in our case typically runs for far fewer instructions, rather than the entire program—the extra time for profiling and a two-phase translation process cannot be amortized. QEMU [2] is the closest dynamic translator to the one described in our work. However, it is optimized for system emulation and our binary translator is optimized for the migration use case.

3. Overview of Migration

When migration is requested (by some entity), the operating system must perform four actions to facilitate migration. It must reschedule the process on another core, change page table mappings to facilitate access to the code for the migrated-to core, perform binary translation until a program transformation point, and transform program state for execution on the new architecture. The first two tasks—process scheduling and page table manipulation—are common operating system responsibilities and require no further discussion here. The final two tasks—binary translation and state transformation—are unique to heterogeneous-ISA migration. We describe these in Sections 6 and 5, respectively.

In this paper, we study in detail a specific instance of the general problem of heterogeneous core migration. In the following paragraphs we describe the language and ISAs/cores we compile for.

We assume that the process to be migrated is written in C, free of inline assembly code. We also assume that externally-linked libraries are compiled for migration (following a similar process to that described in this paper). For libraries that are tuned to specific architectures, this will require some code changes. The C library, for example, because it contains a significant amount of architecture-specific code, will require significant refactoring to be migration-safe. For the testing of our migration technique on programs which are linked to C library code (because we have not yet accomplished the transformation of the libraries), we restrict migration to occur only within non-library code. Finally, we assume some similarities in the ISAs: the same endianness and fundamental data size.

For this study, we model a small, low-power ARM [1] core and a large, high-performance MIPS [18] core. ARM and MIPS are both 32-bit, (typically configured) little-endian, RISC ISAs. However, they represent a significant element of diversity in terms of their memory layout and register access patterns. Diversity in these areas pose more of a problem to migration than diversity of instruction type. The primary contribution of our work is to perform efficient memory transformation during migration, despite this element of diversity.

ARM is most commonly used in highly power-constrained devices, and specific microarchitecture implementations of the ARM core typically reflect this emphasis. MIPS cores, while also commonly used in power-constrained devices, have also been used in more performance-focused devices (like the Origin series of SGI supercomputers). Fundamental differences between these ISAs are discussed in Section 6.2.

We experiment with only two ISAs, but our migration technique should extend in a very straightforward manner to more ISAs on a single CMP. Because we do not create a direct mapping of program state between binaries of each ISA, but rather to a shared compiler front-end intermediate representation, the amount of compiler meta-data used in migration is linear with the number of ISAs.

Fast migration calls for minimal state transformation. This hinges on memory image consistency—the memory image at a point P in the execution of a program on ISA A should be nearly identical to the memory image at P on ISA B so that very little state must be transformed to a machine-dependent form during migration. This is the heart of our migration strategy. The next section describes how memory image consistency is achieved, and the sections following that describe the mechanics of migration, including both the state transformer and the binary translator.

4. Memory Image Consistency

To facilitate fast execution migration, each binary representation of a program (compiled for different ISAs) should expect to find each item of program data at the same virtual address. This makes it possible to perform the migration without having to rearrange data items in memory, greatly reducing the latency of the migration process. In prior systems targeting inter-machine heterogeneous migration, state copy dominates migration overhead and the cost of reordering and transforming memory is inconsequential. Migration on a CMP does not involve memory state copy, so keeping transformation overhead (now the dominant cost) low is critical to good performance.

To achieve memory image consistency, within each section the number of objects, their sizes, their relative order in memory, and their alignment and padding rules must be identical in order for their virtual addresses to be consistent across ISAs. This not only applies to data sections, but also to sections containing code—function definitions must begin at the same virtual address so that function pointers will be correct after a migration. The dynamic portions of memory—the heap and the stack—also need to be consistent.

Although this would best be done by a single compiler with two backends, for this initial study we use two GCC-based compilers and merge the results. Once the compilers agree on memory image layout, we can create a single binary composed of two coordinated text sections and a single version of each compiled data section (since the static data sections created by the two compilers are identical).

4.1 Global Data Consistency

ISAs may utilize entirely different sections for the same data. This is the case for the *small* data sections in MIPS: the *.sbss*, *.sdata*, and *.scommon* sections. Data objects below a certain size are placed in the small data sections; this allows for faster access to these data objects since it takes fewer instructions to reach these items from a base pointer (like the global pointer). This is a consequence of the limited amount of space (16 bits) in a MIPS instruction to specify the offset from the base pointer. ARM overcomes this problem, primarily, by supporting PC-relative addressing. To avoid copying data in or out of small data sections at migration time, we add support for small data sections to the GCC ARM back-end. However, we do not change the compiler to generate ARM code that can take advantage of the small data section by generating code to access data in these sections with fewer instructions—so the performance of ARM code is not improved, but is not degraded either.

4.2 Code Section Consistency

Another program section that must be modified to reduce migration cost is the code (*.text*) section because fixing function pointers during migration is expensive. To eliminate the cost of finding and fixing all function pointers, function bodies should be placed at identical virtual addresses across each ISA. This implies that the order of function definitions in memory be identical. The compiler, therefore, emits function definitions in the order they are encountered in the source files. The size of each function must also be identical; so the assembler adds NOP instructions, when necessary, to pad functions to the appropriate size. While this increases code size a little, we measure no performance loss.

Note that while each function body begins at the same virtual address across binaries, the location of each function call *site* is not identical. Since the address of the call site can vary, the return address will vary—the cost of translating return addresses is incurred at migration time because the constraint of placing call sites at consistent addresses would be too limiting for the compiler and would impact performance.

4.3 Heap Consistency

Code that accesses the heap must be consistent—we call this *heap consistency*—but for different reasons. Heap consistency is not necessary because of pointers—pointers to the heap will work correctly after migration because the addresses of heap objects in the shared space are created dynamically, not hard-coded into the binary. Instead, heap consistency is necessary to ensure that after migration the program has an accurate record of what heap memory is allocated and what is free. This requires that the same implementation of *malloc* be used for all ISAs. Then, at migration time, because all *malloc*'s internal data structures are preserved, a consistent view of the heap will be maintained. The same principle applies to any memory management library a program uses.

In the Linux system that we model, *malloc* acquires memory on behalf of the caller in two ways: through the *brk* system call, which grows the heap, and through the *mmap* system call. The *mmap* system call does not return heap memory, but pages of virtual memory that the operating system has allocated to the process. Since a single operating system instance governs all the cores, a common page table is used, and page allocations will not change, despite the migration.

4.4 Stack Consistency

Objects on the stack may need to be adjusted or reordered for two reasons. First, many instructions have hard-coded stack offsets. Either these instructions must be changed during migration, the objects must be moved during migration, or the objects must be consistently placed by the code (compiled for each ISA) in the first place—we implement a combination of the latter two. Second, some stack objects may have pointers to them; these pointers would need to be fixed during migration if the objects are not placed at identical addresses.

The stack is especially difficult to make consistent without sacrificing performance because stack interaction is carefully optimized for each ISA. For example, in an ISA with a large number of registers, many function arguments may be passed through registers to avoid loads and stores to stack memory. But for an ISA with a small number of registers, most function arguments must be passed on the stack. Our goal is to find the right balance between good runtime performance and low migration overhead. We must change stack memory as little as possible at migration time without eliminating performance-critical ISA-specific stack optimizations. This section describes what we change (and what we do not change) in the stack organization to strike this balance.

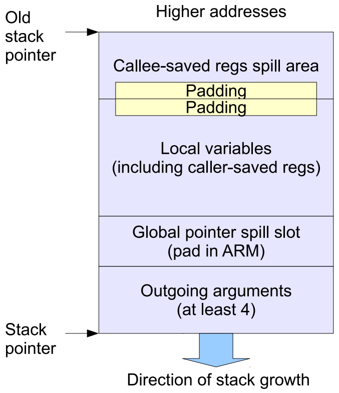
Higher addresses
Old stack pointer
Callee-saved regs spill area
Padding
Padding
Local variables (including caller-saved regs)
Global pointer spill slot (pad in ARM)
Outgoing arguments (at least 4)
Stack pointer
Direction of stack growth

Figure 1. The organization of a non-leaf stack frame in a migration-capable program.

To avoid major changes to the stack during migration, the following properties of the stack must be made identical across ISAs: (1) direction of growth, (2) size, (3) ordering, and (4) alignment. First, the stack's direction of growth should be the same: in both ARM and MIPS, the stack grows downward. Also, for every function in a given program, we make the stack frame size the same for each ISA (by adding padding), making the overall stack size consistent. It is also necessary to have a consistent ordering of regions within each frame. The cross-architectural layout of a non-leaf stack frame is shown in Figure 1. In order to achieve identical frame size, it is also necessary to maintain consistent alignment and padding rules between regions. In our implementation, each region is 8-byte aligned.

In addition to the changes to the overall structure of the stack, we must make changes to each major frame sub-component to avoid costly stack transformations during migration. The major frame sub-components are: (1) the function arguments region, (2) the callee-saved register spill area, and (3) slots for local variables.

Function Arguments. After migration, the program must be able to locate arguments to open functions—we desire to do this with minimal transformation during migration. We recognize that forcing arguments on the stack that would otherwise have been passed through registers can hurt performance, so we are careful to avoid such a change. Both ARM and MIPS pass the first four arguments through registers, but MIPS also allocates stack space for these arguments in case the callee needs to spill them. We force ARM to do the same—space is allocated for the first four arguments, though the arguments are always passed through registers. This increases memory usage by about 16 bytes per frame; however we observe no noticeable effect on performance.

Callee-saved Register Spills. For a given function, the number of callee-saved registers that need to be overwritten depends heavily on the number of registers the ISA supports (affecting register pressure), constraints placed on ISA-specific special-purpose registers, and low-level code transformations. Because all of these factors are architecture-specific, the size of the callee-saved register spill area differs greatly across ISAs. We therefore modify the compiler to add padding, as necessary, to the callee-saved register spill area of each stack frame. This change has no noticeable effect on performance.

Local Variables. The last frame sub-component that requires modification is the area reserved for local variables. Our changes ensure consistent ordering and identical size (requiring padding). For MIPS, GCC allocates variables in this region from low address to high; for ARM from high address to low. Large aggregate objects (like structs and arrays) and objects whose addresses are taken are allocated to the stack first. The different allocation directions cause these objects to be allocated at different addresses on the stack. For pointers to these objects to work after migration, we require all of these objects to be allocated to the same addresses across ISAs. To enforce this requirement we change the allocation direction; when GCC compiles ARM code it now allocates stack objects from low address to high. Like the other regions of a stack frame described above, this region may require padding because the number and choice of local variables to allocate depends on the number of registers in the ISA, resulting in different frame size.

5. Migration Process

When all of the compiler/assembler/linker changes described in the previous section are applied, all program sections, except for some portions of the stack, are migratable without any transformation. To facilitate the transformation of a process' memory image (specifically, the stack portion) we introduce a small program called the Stack Transformer (ST). The ST has three jobs: (1) It creates the register state for the migrated-to core. (2) It fixes all the return addresses on the stack. (3) It moves the values of local variables in open function activations to the right stack offsets. The ST must ensure that the value of every stack-based variable is at the address where code for the migrated-to core expects it.

The Stack Transformer (ST) needs detailed information about the compilation of the program from source code in order to prepare the stack for execution on the migrated-to core. We modify GCC to record the necessary information during compilation. This information includes the frame layout for each function, details about function call sites, the locations of local variables, and the sets of spilled caller-saved and callee-saved registers. This compilation meta-data can easily be placed in a new ELF section and bundled with the binary (in the same way debugging information is) to make the executable self-contained.

5.1 Optimizations that Interfere with Variable Location

A few late optimization passes in GCC expose new live registers across call sites. They do so in such a way that these new live registers do not always correspond to a high-level variable name; moreover, the transformations are not always applied uniformly across all architectures, resulting in architecture-specific state that cannot be transformed at migration time. The incompatibility of these passes with migration is not inherent to the optimization algorithms, but an artifact of their implementation in GCC. To make these optimizations compatible with migration, one of two things may be done. First, the optimizations can be converted from RTL passes to tree passes because the tree intermediate representation is architecture-independent. In this way, optimization can be applied consistently during compilation for both architectures, and intermediate state that is exposed can be assigned labels that are carried in the intermediate representation until register assignment. Second, the optimization passes can be modified to avoid applying the optimization across call sites—we do this for the common subexpression elimination pass. How this is to be done depends on the operation of the optimization. If the optimization moves code, then code motion across call sites should be avoided. If the optimization deletes instructions, it should avoid deleting an instruction that exposes a new live register at any call site.

Both solutions—converting RTL passes to tree passes and introducing optimization barriers at call sites—may result in less than ideal performance over the current implementations. But given the minimal performance degradations we observe with these optimizations completely disabled, we expect the performance tradeoff to be minimal and acceptable. Either solution would require a significant refactoring of the code; so for this study, we have chosen to disable the problematic RTL optimization passes, including loop invariant motion, global common subexpression elimination, forward propagation, post-reload instruction scheduling, temporary expression elimination, and tail call elimination. Disabling most of these passes results in negligible performance loss. For the benchmarks we use in this study, we observe some performance degradation when the following passes are disabled: global common subexpression elimination (less than 1%), forward propagation (less than 1%), post-reload instruction scheduling (less than 0.5%), and temporary expression elimination (less than 1%). Turning off all these passes results in a performance loss of 3.1% in ARM code and 1.6% in MIPS code.

A compiler designed from the ground up to emit multiple-ISA code would not have these problems, and these passes would not have to be deactivated. In that case, we believe the performance cost of restricting a few optimizations across call sites would be close to zero.

5.2 Operation of the Stack Transformer

The job of the Stack Transformer is to transform the architecture-specific program state (mainly stack data, but also register state) from ISA A to ISA B, so that the program running on ISA B will find all of its data after migration where it expects it. At migration time multiple functions may be open at once (represented by a sequence of frames on the stack). The value of each local variable in each open function may be in one of three locations: a register, a fixed stack slot allocated on function entry, or a register spill location on the stack (either in the owning function's frame or in the frame of a descendant function). The code compiled for the destination core will likely expect the value of each variable at a different location (due to differences in the type and number of registers). For example, register pressure may have forced the compiler to allocate variable 'x' to stack offset 20 in code for ISA A, but code for ISA B (which has more registers) put 'x' in register 18. The ST must move the values of all live variables in all open functions to the locations expected by the code for the destination core. It walks the stack, one frame at a time, locating and repositioning live local variables.

From a high level, the ST performs two quick passes over the call stack. Two passes are necessary because we do not know which variable's value should be placed in a callee-saved spill location until we discover the ancestor function that keeps one of its local variables in the callee-saved register. The first pass goes from innermost frame (the frame in which execution was stopped) to outermost frame and finds values for caller-saved spill locations and simple stack-bound variables. The second pass works in the reverse direction—from outermost frame back to innermost frame—finding values for callee-saved spill locations and determining final register state. We describe each of these passes in turn.

First Pass. The first pass examines and transforms each stack frame from the most recently opened frame to the first frame created. Before it can start locating live local variables, the ST must discover which function execution stopped in, and which call site it stopped at (since different variables may be live at different call sites). Therefore, when the ST begins, it is given the value of the PC on the source core. Using the PC, the ST looks up information about the call site (recorded during compilation). The ST then looks up information on the function containing the call site (also recorded during compilation). Finally, the ST looks up records for the same call site (using the call site UID) and function for the other ISA (the ISA of the destination core). With these four records—call site information and function information for each ISA—the ST goes to work transforming the stack frame.

First, it processes the list of registers that are live across the call site. It looks for the values of these registers (by variable name) in three places: (1) live registers on the source core, (2) caller-saved spill locations, and (3) fixed stack slots. When checking fixed stack slots, associated scope information for each variable may be used; the scope of the call site must be contained within the scope that the variable is defined in. Once the ST has located the value that the register should have at the current call site, it copies the value to a list of live register values for the current function. It will keep such a list of live register values for every frame encountered, to be used later in the reverse call stack traversal. In some special cases, the value for the live register will be a constant (which would have been recorded during compilation) or the address of a global variable (which can be looked up in the program's symbol table).

Next, the ST finds the values of variables in the caller-saved spill slots and variables at fixed locations on the stack. The ST checks the same three sources that it used to find values for the live registers on the destination core. Only small variables whose addresses are never taken are moved. Large variables on the stack (like arrays and structs) and variables whose addresses are taken are given identical stack locations by the compiler, so no copying of these values is needed, and pointers to these variables will remain valid after migration.

When the ST is done processing a frame it will determine the live register state on the source core as it existed immediately before the current function was entered. It will carry over this register state when it transforms the next frame. It updates its snapshot of the register state on the source core by reading from the callee-saved register spill locations in the current frame.

Besides moving local variables to their expected locations, the ST is also tasked with fixing all the return addresses saved on the stack. Each stack frame stores the return address of the call site of its caller. Because only function heads, not call sites within functions, are placed at identical addresses by the compiler, return addresses need to be fixed so that return instructions that are eventually executed on the destination core will work properly. When the ST visits each frame to move local variables, it also fixes return addresses. Since the compiler records the size of each frame and the offset where the return address is saved, the ST can find the saved return address. The ST looks up compiler-recorded data on the parent function and repeats the transformation procedure for the next frame up the call stack.

Second Pass. After the ST has passed over all the stack frames from innermost to outermost, it has enough information about the live register state at each call site to determine the values of the important callee-saved spill locations. The ST starts with live registers in the outermost frame (which it recorded in the previous pass) and moves to the next (inner) frame. For each callee-saved register, it copies the value from its snapshot of the current register state to the appropriate location on the stack. Then it updates the current register state with the live register values at the next call site (which it recorded in the previous pass). The ST moves to the next frame and repeats the process until all the important callee-saved spill slots are full. The register state at the end of this process is the register state that should be instated on the destination core when the ST is done and the core is ready for native execution of the program.

There are two important things to note about this process. First, it handles the case where a live register at a particular call site is not saved immediately by the callee (because the callee doesn't recycle that register), but is instead saved in some distant frame or is never saved—with the value remaining in the register at the time of migration. This would not be possible with only one pass from innermost to outermost frame. Second, if the callee spills a register that is not live at the call site, the ST may not have a value to place in that spill slot; but this does not cause any problems after migration because when the called function is returned and the register is filled, it is not live, so it will not be read before being overwritten.

6. Binary Translation

Binary translation is performed on a migrated process until it reaches an equivalence point, at which point the stack transformer described in the previous section transforms program state for native execution. Our binary translator uses the following scheme of classic just-in-time (JIT) [7] dynamic translation:

- Starting from the instruction at the point of migration, each instruction is translated to the ISA of the migrated-to core and placed in a code cache until we encounter a function call site or an indirect/conditional jump instruction (whose target address is not known until execution).
- Next, a stub (a short group of additional instructions) is added to the end of this translated block of instructions. The stub contains a jump to the stack transformer if we've reached a function call site. Otherwise, the stub saves the target address at a known location and jumps to a translator core function called the *translation engine*.
- Control is then transferred to the translated code in the code cache. If the code eventually relinquishes control back to the translation engine, we repeat the above steps from the instruction at the target address until we finally reach a function call site.

6.1 Translation Block Chaining

The translation engine, before translating the next block of instructions, checks if the block is already available in the code cache. If it is available, it links the end of the previous block to the beginning of the next block, with a direct branch instruction. This process is known as translation block (TB) chaining and has been extensively applied in emulators and virtual machines [22].

We extend this idea by allowing translation block chaining from any instruction in the middle of a TB to any instruction in another TB. This allows for a TB to be chained to more than one TB at different instructions (the most common case is a conditional branch). For example, TB *X* can be chained with TB *Y* at *X.i*, with *Z* at *X.j* and with itself at *X.k*, where *X.i,j,k* represent three different instructions in X. The converse is also true: TBs *X* and *Y* can both chain to *Z* at *Z.i* and *Z.j* respectively. In this case, however, *Z.i* and *Z.j* can be the same. Merge point is a classic example for such a scenario, wherein the "if" part can be in TB *X*, "else" part in TB *Y*, and they both chain to TB *Z* at their merge instruction *Z.i*.

We call this *Multiple-Entry Multiple-Exit (MEME) translation block chaining*. The following issues need to be addressed by such a design:

Condition Codes. In a MEME TB, any instruction can have multiple entry points, including those that check condition codes. To improve performance, our binary translator performs lazy condition code evaluation, which defers evaluation of a condition code until it is checked. Any instruction that checks a condition code (CC) evaluates it first, if it has not already been evaluated by an instruction prior to that in the same TB. With MEME chaining, we can never be sure whether a CC has been evaluated or not, due to multiple entry points. Also, we do not know which instruction modified the CC in the first place, to perform lazy evaluation accordingly. To overcome these issues, we update a dirty CC map register at every exit point. The dirty map can be used by instructions to check if a CC has been already evaluated. In addition to this, we also store the opcode of the last CC modifier instruction in a register, so that at the time of lazy evaluation, we would know which instruction modified the CC.

Program Counter Updates. ARM allows instructions to use the program counter as a general-purpose register. For performance reasons, the (virtual) program counter is not updated after executing every block of target instructions that emulates a source instruction. It is instead updated whenever necessary. We further optimize this by adding an offset from the last-calculated PC rather than moving the entire 32 bit address (32 bit move takes at least two MIPS instructions). With MEME chaining, the last-calculated PC can be different for different entry points. To overcome this, we maintain a map of the last-calculated PC at every instruction in a TB. Using this map, the last-calculated PC is updated at the end of every exit point.

Instruction Scheduling. Instructions within a translation block might be reordered. We do not do instruction scheduling optimizations, but we do fill branch delay slots. Branch delay slots are filled with an independent instruction prior to the branch in the same TB. This is not always correct if the branch has multiple entry points. We handle branch delay slots in ARM to MIPS translation as follows:

- All register indirect branches trap into the translation engine.
- All direct branches are evaluated by the translator, which translates instructions at the target address and inserts them inline. If they are already translated, we do MEME chaining with the lazy PC update instruction in the branch delay slot, which has to be executed regardless of the entry point.
- All conditional branches are only dependent on condition codes which are taken care of by lazy CC evaluation. So the instruction just before the branch is not dependent on the branch and hence always goes into the delay slot. When MEME chaining happens at a conditional branch, we move back any instruction in the delay slot and insert a NOP into the delay slot instead. Performance reduction due to this is negligible compared to the significant gain in performance due to TB chaining.

Delay slots are not a problem in MIPS to ARM translation because ARM does not have delay slots.

As a further optimization, we have the ability to preserve the code cache across migrations, so that if the process is migrated to the same core type again, there is a good chance that the code we need is already present in the cache and can be directly used. However, we do not employ this optimization in our presented results.

6.2 ISA-Specific Challenges

Despite high-level similarities, ARM and MIPS represent significant diversity. ARM has many features that MIPS lacks: condition codes and an abundance of predicated instructions, load multiple and store multiple instructions, a program counter that is accessible as a general-purpose register, and finally PC-relative load instructions to access data embedded in the text section. MIPS, on the other hand, has double the number of integer registers that are accessible to programmers and allows for 16-bit immediates as opposed to the 8-bit immediate restriction in ARM. Finally, each ISA has a different *application binary interface* (ABI); they use different system call numbers and follow different conventions to make system calls. We discuss several of these challenges in this section.

Register Allocation. *Mapping from ARM to MIPS:* ARM has 16 general-purpose registers visible to the programmer, while MIPS has 31 general-purpose registers (excluding R0). Hence, all 16 registers in ARM are easily mapped to registers in MIPS. In addition, we reserve four MIPS registers for the ARM condition codes (Zero, Negative, Carry, and Overflow) and an extra register for inverse carry. Of the remaining 10 registers, four are used for lazy condition code evaluation as described below, three are used as temporary registers, and the remaining three are reserved for future use.

Mapping from MIPS to ARM: All 31 MIPS general-purpose registers cannot be mapped onto registers in ARM. Hence, we map frequently used MIPS registers (R1–R7, global pointer, stack pointer, frame pointer and link register, collectively called the "mapped" registers) to registers in ARM. One register points to an in-memory register context block where the "unmapped" MIPS registers are placed. In addition to this, we reserve three registers as cache registers that contain the three most frequently used unmapped registers for faster access.

Condition Codes and Predicated Instructions. Most translators use a global data flow analysis technique to perform a lazy evaluation of condition codes. However, since we perform binary translation for a relatively small number of instructions until we reach an equivalence point, a data flow analysis would increase migration overhead significantly. Hence, we use a lazy condition code evaluation scheme similar to the one used in QEMU [2]: for every instruction that updates a condition code, we store the opcode, operands, and result in temporary registers reserved for the lazy evaluation, and compute the condition codes using this information whenever required. In addition to this, a dirty map register is used to support MEME TB chaining as described above.

Once the condition codes necessary for a predicated instruction are evaluated, a branch instruction is used to test the condition, which skips the operation performed by the instruction if the condition is false. The conditional move instruction in MIPS is used to translate conditional move instructions in ARM, but we use a conditional branch around arithmetic instructions for more complex predicated instructions.

Immediate Instructions. MIPS restricts the size of immediates to 16 bits while ARM limits them to 8 bits. This necessitates two ARM instructions to construct a MIPS immediate, store it in a register, and then perform the actual operation. The problem worsens with 32 bit immediate updates like link register or program counter updates, where four ARM instructions are needed to perform the move. To overcome this, we extend the register context block to also include a "cache of immediates", so that one load instruction will suffice for the entire operation, as opposed to the four mutually dependent shift-OR instructions.

System Calls. In this work, we assume a single operating system instance running on both the cores. This guarantees that a system call works in the same way on both the cores. However, the system call numbers and calling conventions are dictated by the ISA's ABI, which requires a remapping step when in binary translation mode.

One approach would be to provide system call emulation during translation. As an alternative, it is a minor change to our system to also make system calls equivalence points, like function calls. This eliminates the need for system call emulation, but also increases the frequency of equivalence points, which is also a useful feature. To support migration while executing a system call, we would need to apply our techniques and methodology to the operating system itself.

7. Experimental Methodology

To test migration and measure the runtime performance effects of compilation for migration, we use the SPEC2000 Integer benchmarks written in C (i.e., all but the C++ benchmark, *eon*). From this set of benchmarks we exclude the *gcc* benchmark because it uses the non-standard *alloca* C library function to dynamically allocate memory on the stack instead of on the heap—at this time our migration technique does not support variable-size stack frames. All benchmarks are compiled with GCC at optimization level -O2. In all simulations the reference inputs are used. We use the M5 processor simulator [3] (configured to execute ARM binaries and MIPS binaries). The architectural model of the ARM core is based on the low-power Cortex-A8 core, while the MIPS core is modeled with

ARM core			
Frequency	833 MHz	I cache	32 KB, 4 way
Fetch/commit width	2	D cache	32 KB, 4 way
Branch predictor	local	L2 cache	2 MB, 8 way
MIPS core			
Frequency	2 GHz	I cache	64 KB, 4 way
Fetch/commit width	4	D cache	64 KB, 4 way
Branch predictor	tournament	L2 cache	4 MB, 8 way

Table 1. Architecture detail for ARM and MIPS cores

performance as the primary design objective. The details of each core are given in Table 1.

To evaluate the steady-state behavior of each benchmark on each core type, we simulate a portion of execution for each benchmark. To ensure that we measure the performance of the modeled cores on the same unit of work even after benchmark recompilation, we insert two marks in the source code of each benchmark—one to indicate where detailed simulation should start and one to indicate where it should stop. The first mark is made at the point in the code after approximately one billion instructions have passed (to skip over initialization code) and the second mark is made after approximately 500 million more instructions have passed. So the simulation interval is approximately 500 million dynamic instructions, the exact number of dynamic instructions depending on the ISA and compilation options.

To evaluate the average cost of migration, we cross-compile both the Binary Translator and the Stack Transformer for ARM and MIPS and run them in the M5 simulator on sample migration points taken from the benchmarks. For these experiments, we venture further into the executable than in the steady-state experiments, to gather more samples of potential migrations points. We collect 10 samples for each benchmark in each direction of migration. The samples are collected at intervals of 100 million dynamic instructions starting one billion instructions into execution. We first run the binary translator on each sample, and perform stack transformation as soon as execution reaches the nearest function call site. In a few cases migration at the nearest call site is not possible due to issues like executing in migration-unsafe library code; so in those cases, a nearby sample that is suitable is used.

The Stack Transformer is written in C++. Designed as a proof-of-concept, there is still plenty of room for performance optimization, and the performance results for the ST presented in this paper should be considered conservative estimates of potential performance.

8. Results

The goal of this research is to enable multi-ISA heterogeneous architectures. Such an architecture will be most useful if we can migrate threads often enough, with low enough overhead, to actually exploit that heterogeneity within a single process execution. This section outlines the costs of migration, including the runtime (migration-free) cost, the stack transformer, and the binary translator.

8.1 Runtime Performance of Migratable Code

A key goal of this work is to enable fast migration without compromising runtime performance—that is, performance when no migration is occurring. Throughout the toolchain changes to ensure a nearly identical memory image across architectures, care must be taken that performance is not compromised. Among our benchmarks no performance is lost due to changes to make the memory image consistent, including the addition of padding (which is too little to cause more instruction cache misses).

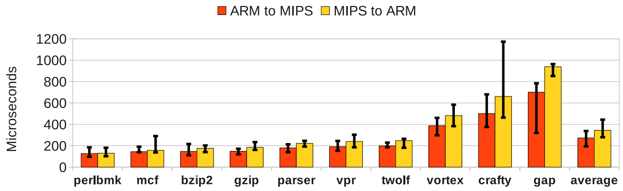

Figure 2. The average costs of state transformation for migration. Lines indicate the minimum and maximum measured transformation times.

There is, however, some performance loss due to optimizations in GCC that are disabled because they inconsistently move code across call sites (that is, call-crossing code motion is applied in only one ISA) and/or they impair the association of variable names with locations (preventing the ST from being able to move some data values). The specific changes that result in performance loss are described in Section 5.1. On average, runtime performance only suffers by 3.1% in ARM code and 1.6% in MIPS code. Again, these are an artifact of a compilation system not intended to create dual executables, and we expect these costs would disappear in a native dual-ISA compiler.

8.2 Migration Cost

The cost of performing a migration is a combination of binary translation overhead, overhead from the involvement of the operating system, and the execution time of the Stack Transformer. Since we do not have an OS for a heterogeneous-ISA CMP, we focus only on the migration cost incurred by binary translation and the Stack Transformer. However, we expect the incremental OS overhead to be relatively small, since its role is to add the thread to the run queue of another core and change some page table entries (to swap out code sections).

For our benchmarks, state transformation (i.e., ST execution time) on our detailed architectural models for ARM and MIPS cores takes (on average) 272 microseconds for migration from ARM to MIPS and 344 microseconds for migration from MIPS to ARM. Figure 2 shows the average transformation costs for each benchmark for migration in each direction. It also shows the fastest and slowest transformation times we measure, as error bars. We do not compare the overhead of the ST (which, due to compiler support, has very little to transform) to a naïve transformer (that must transform most of the memory image, including all pointers) because it would be several orders of magnitude slower.

For migration in either direction, the transformation overheads for the benchmarks *vortex*, *crafty*, and *gap* are the greatest. These three benchmarks also have the highest average stack depths. The average stack depth is the average number of frames on the call stack at any given point in execution. *Gap* has an average stack depth of just under 30. On the other end of the spectrum, *perl* has an average stack depth of three. The average stack depth across all the benchmarks is nine. Stack depth has the greatest influence on migration cost in our migration technique; the deeper the stack, the more state needs to be transformed.

Figure 3 shows the relationship between performance and migration frequency. The performance results presented in this graph account for both performance costs due to compilation for migration (discussed in the previous section) and transformation overhead. In this graph, migration frequency refers to how frequently migrations take place. For example, a frequency of 20 milliseconds means that every 20 milliseconds the program switches cores.

We assume in this graph that we can migrate at any time—that potential migration points (call sites) occur frequently (Section 8.4

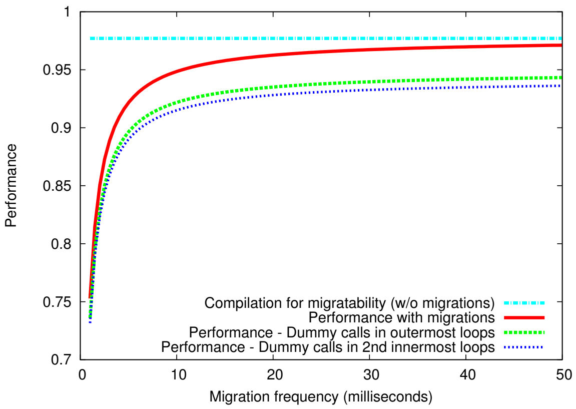

Figure 3. Performance (without binary translation) vs. migration frequency when migrating back and forth between an ARM core and a MIPS core.

quantifies the additional cost of binary translation). The straight line in the graph represents the performance if no migration occurs, and accounts only for the overhead of compilation for migratability. The line below that shows performance when migrations occur. When migrations happen every 10 milliseconds or less the effect on performance is significant. But above 10 milliseconds, performance remains above 95%, as compilation cost dominates migration overhead. The other two lines will be discussed in the next section.

8.3 Frequency of Potential Migration Points

In a system without binary translation, we would have to postpone migrations until the next function call is encountered. Even with binary translation, the latency to reach the next function call site is important because it determines how many instructions must be executed by our binary translator, which incur a performance overhead.

The distribution of function calls is highly irregular. During some phases of execution, function calls are frequent; in other phases, calls are much less frequent. Consequently, the average time between calls and the median time between calls are poor metrics for evaluating call frequency. Instead, we select as our metric the expected time to transformation (ETTT). This is the average time (measured in dynamic instructions) until the next call site is reached from any randomly-selected point in execution.

Figure 4 (first two bars) shows the ETTT for each benchmark. From this graph it is clear that call frequency varies dramatically across benchmarks. Among the ARM binaries, on one end of the spectrum is *vpr* where, on average, one must wait only 84 dynamic instructions until the next possible transformation opportunity. On the other end of the spectrum is *mcf* where, on average, one must wait over 48 million instructions until the next possible transformation opportunity. For seven of the ten benchmarks, the ETTT is under 130,000 dynamic instructions; but for *bzip2*, *crafty*, and *mcf* the ETTT is much higher.

One way to increase the frequency of migration opportunities is to add more function calls. The long gaps between calls result from long-running loops that do not contain any function calls (or have had their function calls inlined). We modify GCC to inject dummy function calls (calls to an empty function that returns immediately) into loops. We experiment with two loop selection policies following two simple heuristics: inject dummy calls in outermost loops and inject dummy calls in second-innermost loops (parents of innermost loops). We never select innermost loops because the performance impact is too severe [11].

The impact on ETTT of these policies is shown in the last four bars in Figure 4. For the three benchmarks with the longest ETTT, inserting function calls in outermost loops dramatically improves call frequency. For example, in the MIPS binaries, the ETTT for *bzip2* drops from over 19 million to about 3.1 million dynamic instructions; for *gap*, the ETTT drops from about 7.4 million to about 4 million dynamic instructions; and *mcf* sees its ETTT reduced from almost 60 million to under 200,000 dynamic instructions. Using the second loop selection policy (second-innermost) results in only marginal gains—most benchmarks only see a slight decrease in ETTT. *Bzip2* is an exception, dropping by a factor of 7.9 on ARM and 4.5 on MIPS.

The additional transformation opportunities that these changes bring comes at the cost of performance. Injecting dummy calls adds instructions that do not do any useful work and may interfere with some compiler optimizations. Performance drops 1.4% on ARM and 4.7% on MIPS when dummy calls are added to outermost loops for all applications. When calls are added to second-innermost loops, performance drops 2.3% on ARM and 5.4% on MIPS. For some programs (ones with infrequent calls) the performance degradation may be justified if the migration policy demands it. Note that if we assume the compiler is smart enough to only apply dummy calls for applications that need them, the cost would be significantly lower.

The two lowest bars in Figure 3 show the performance of code compiled with dummy calls inserted. Migration overhead is the same, but due to the additional compilation costs, performance is lower at every migration frequency. If migration never occurs, performance remains below 95%.

For the results in the next section, we do not assume the presence of dummy procedures calls. This preserves some non-migration performance, but produces very conservative, worst-case numbers for expected binary translation overhead.

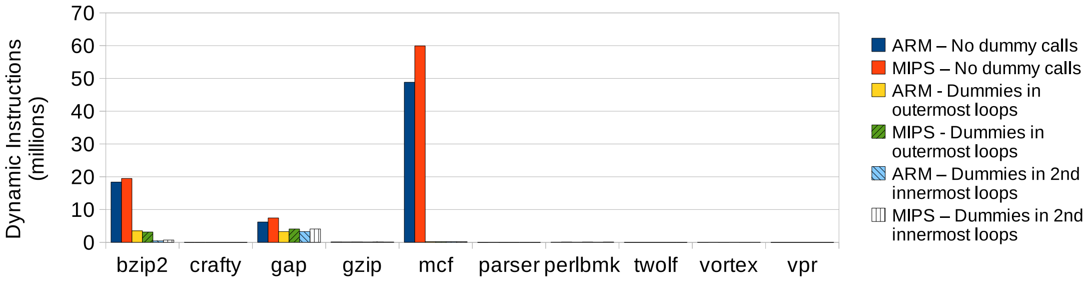

Figure 4. The expected time to the next call, under three situations: no dummy calls have been added, dummy calls to outermost loops have been added, and dummy calls to second-innermost loops have been added.

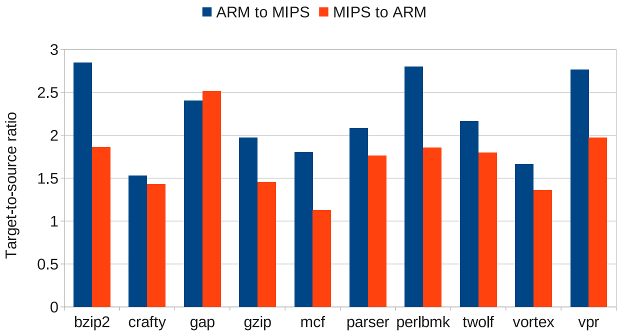

Figure 5. The ratio of the number of target instructions executed during binary translation to the number of source instructions during native execution.

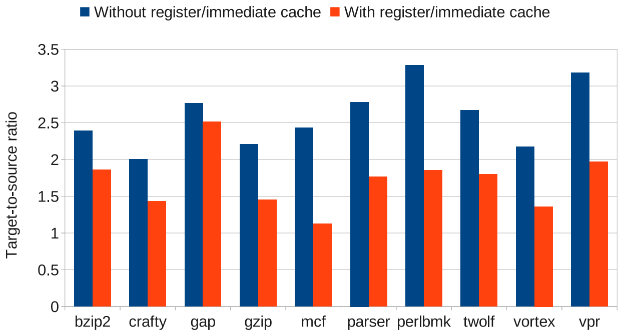

Figure 7. Target-to-Source ratio during Binary Translation from ARM to MIPS—with and without Register and Immediate caches

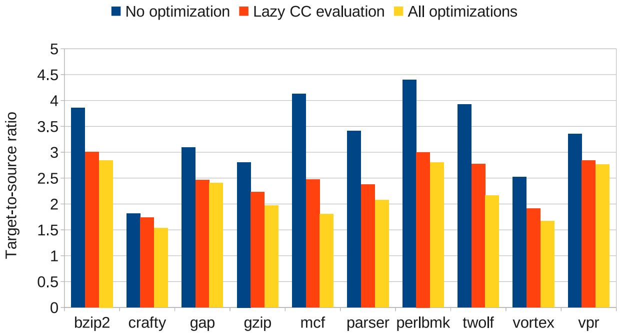

Figure 6. Target-to-Source ratio during Binary Translation from ARM to MIPS—without optimization, with lazy condition code evaluation and with full optimization.

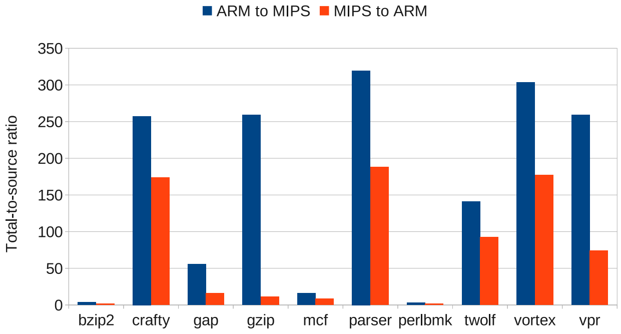

Figure 8. The ratio of the number of dynamic instructions executed (including the ones used for translation) during binary translation to the number of source instructions during native execution.

8.4 Binary Translator Performance

We characterize the performance of our binary translator based on the following three metrics:

- **Target-to-Source Ratio:** The ratio of the number of dynamic target instructions executed on the migrated-to core to the number of dynamic source instructions executed on the native core. Most static binary translators report a target-to-source ratio between one and two. Dynamic binary translators tend to have a higher target-to-source ratio because they have less scope for optimization.
- **Total-to-Source Ratio:** The ratio of the number of dynamic instructions inclusive of both the target instructions and the instructions used for translation (as executed on the migrated-to core) to the number of dynamic source instructions executed on the native core. This is the ratio we address with our translation block chaining algorithms.
- **Overhead due to binary translation:** Time taken for binary translation compared against time taken on native core, in microseconds.

Figure 5 shows the target-to-source ratio for each benchmark in both directions of migration. The target-to-source ratio for MIPS to ARM translation is generally less than that from ARM to MIPS, due to the more complex instructions in ARM. One exception to this is *gap*, which has a higher ratio for MIPS to ARM translation. This is because *gap* is characterized by tight loops with multiply instructions, which takes additional instructions in ARM to perform stores to "hi" and "lo" memory locations. In both directions, *bzip2*, *perlbmk*, and *vpr* have high target-to-source ratios because of a large number of branches in MIPS code and large number of predicated instructions in ARM code.

Figure 6 shows the target-to-source ratio for ARM to MIPS translation without optimization and then with each optimization applied incrementally. The leftmost bar shows the performance of a naïve binary translator without any optimizations. The middle bar shows the performance of the binary translator doing lazy condition code evaluation. This optimization significantly reduces the dynamic instruction count. The rightmost bar shows the binary translator with all optimizations enabled. This includes grouping predicate instructions and certain constant-folding optimizations. Lazy condition code evaluation contributes the most to the overall translation speedup.

Figure 7 shows the performance of the MIPS to ARM translator with and without the use of the register cache and immediate cache. Dynamic instruction count is significantly lower with these optimizations. Our register cache is made up of only three temporary registers. We expect that a larger cache with an adaptive register allocation strategy should give even greater speedups.

Figure 8 shows the total-to-source ratio for each benchmark in both directions of migration. Again the MIPS to ARM translator has a lower translation cost than ARM to MIPS, because it does not have to make complex decisions like lazy condition code evaluation and lazy PC update during translation. However, there is a high irregularity in the total-to-source ratios of different benchmarks—

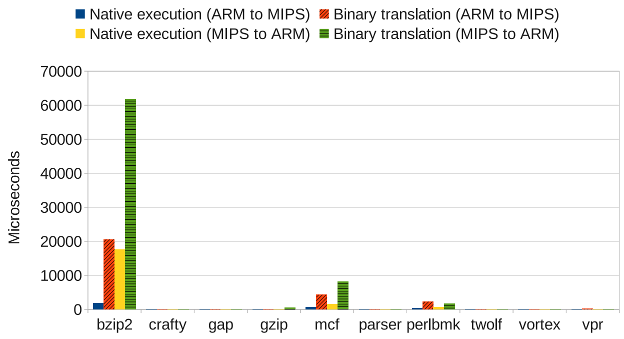

Figure 9. Comparison of Binary Translation time with Native Execution time in microseconds

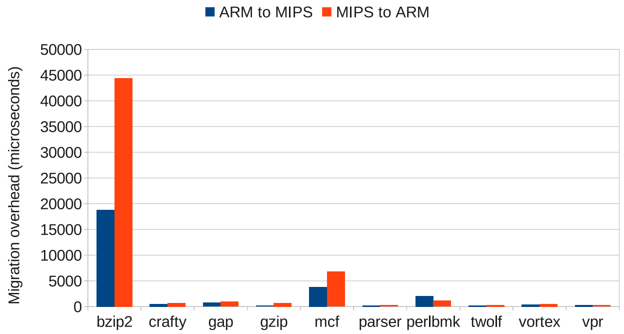

Figure 10. Migration Overhead due to Binary Translation and Stack Transformation in microseconds

Benchmark	ARM to MIPS	MIPS to ARM
bzip2	99.9992	99.993
crafty	0.0	0.0
gap	85.9	94.0
gzip	18.1	99.9
mcf	99.96	99.1
parser	15.7	30.7
perlbmk	99.997	99.99
twolf	58.9	65.4
vortex	0.0	0.0
vpr	36.6	67.1

Table 2. Percentage of instructions used from code cache

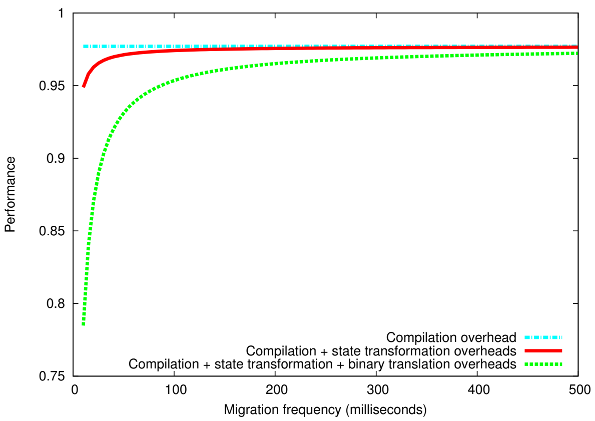

Figure 11. Performance vs. migration frequency when migrating back and forth between an ARM core and a MIPS core. Performance includes overheads due to compilation for migratabiltity, state transformation, and binary translation.

some are as high as 300 while some are close to one. This is heavily impacted by the number of instructions to the next equivalence point (call site). If it is 84 instructions (the average for *vpr*), the cost of translation is not amortized. If it is hundreds of millions of instructions, the vast majority of execution is in code cache and the translation cost is insignificant. The latter is due, in large part, to the MEME chaining which allows execution to remain in the code cache once a steady state is reached. Table 2 shows the percentage of dynamic instructions executed in the code cache during binary translation.

We next compare the performance of our binary translator with native execution in Figure 9. Within the migration points we sample, the average execution time for an ARM binary on an ARM core from the time migration is requested until an equivalence point is reached is 284 microseconds, while the average binary translation time of the ARM binary on a MIPS core is 2745 microseconds. For ARM to MIPS, the average execution time on a MIPS core is 1981 microseconds while binary translation time on an ARM core is 7240 microseconds. Thus, binary translation costs us 2461 microseconds while migrating from ARM to MIPS and 5259 microseconds while migrating from MIPS to ARM. All benchmarks except *bzip2*, *perlbmk*, and *mcf* complete binary translation in tens or hundreds of microseconds. These three benchmarks show a high binary translation overhead because they have to translate millions of instructions before reaching an equivalence point.

Finally, we evaluate the performance of our migration strategy by looking at the total migration overhead—time taken by both binary translation and stack transformation. This is shown in Figure 10. The average migration overhead for ARM to MIPS migration is 2734 microseconds, while it is 5602 microseconds for MIPS to ARM migration. It should be noted that this average is dominated by a few outliers. If we ignore *bzip2*, *mcf*, and *perlbmk*, the average overhead drops by about a factor of 10.

Figure 11 shows performance (relative to native execution without migration) at different migration frequencies (when migrating back and forth between cores at fixed time intervals), assuming average cost values. It breaks down the costs due to compilation for migratability, state transformation, and binary translation. With all costs considered, execution is 95% as fast as native execution when migration occurs, on average, every 87 milliseconds. This would be an extremely high rate of migration for most foreseeable applications. Also recall that much of that 5% lost performance is an artifact of the compiler not being designed from the ground up to emit multi-ISA code.

We believe this level of performance crosses a critical threshold. Unless the code is migrating between cores nearly every timer interrupt, the cost of migration is very small. This means that the difference in migration cost between a single-ISA heterogeneous CMP and a multi-ISA CMP is negligible for most reasonable assumptions about desired migration frequency. Thus, there is no significant performance barrier to fully exploiting heterogeneity in a multicore architecture, including both microarchitecture heterogeneity and ISA heterogeneity. For comparison, recall that prior work typically measured migration time in hundreds of milliseconds [11], if not worse, and that migration could not occur at an arbitrary point in execution.

9. Conclusion

In this paper we present a new technique for execution migration in a heterogeneous-ISA CMP. This environment affords a unique opportunity for fast migration because migration overhead is not dominated by state copying since memory is shared among all cores. Our migration technique takes advantage of this opportunity by compiling programs to maintain memory state in a way that is nearly identical to its representation on every core type. As a result, migration requires only minimal transformation—only portions of the stack and register state need to be transformed. All pointers remain valid after migration without any transformation, eschewing the need for expensive pointer fixing during migration. We demonstrate that with strategic compiler changes, runtime performance does not have to be compromised for migratability—on average, non-migration performance is reduced by 1.6% for MIPS and 3.1% for ARM. The state transformation cost for migration is, on average, 272 microseconds for ARM to MIPS migration and 344 microseconds for MIPS to ARM migration. We support migration at all points of execution in the program with the help of binary translation. We incur an average binary translation cost of 2.75 milliseconds for ARM to MIPS migration and 7.24 milliseconds for MIPS to ARM. Finally, we show that even if we were to migrate every few hundred milliseconds, we experience a total loss in performance of well under 5%.

Acknowledgements

The authors would like to thank the anonymous reviewers for their helpful insights. This research was supported in part by NSF grant CCF-1018356.

References

[1] ARM Limited. *ARM Architecture Reference Manual*.

[2] F. Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Technical Conference*, Apr. 2005.

[3] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt. The M5 simulator: Modeling networked systems. *International Symposium on Microarchitecture*, Dec. 2006.

[4] G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill. Automated application-level checkpointing of MPI programs. In *Symposium on Principles and Practice of Parallel Programming*, June 2003.

[5] C-Port Corp. *C-5 Network Processor Architecture Guide*.

[6] J.-Y. Chen, W. Yang, T.-H. Hung, H.-M. Su, and W.-C. Hsu. A static binary translator for efficient migration of ARM-based applications. In *Workshop on Optimizations for DSP and Embedded Systems*, Apr. 2008.

[7] L. P. Deutsch and A. M. Schiffman. Efficient implementation of the smalltalk-80 system. In *Symposium on Principles of Programming Languages*, Jan. 1984.

[8] F. B. Dubach, R. M. Rutherford, and C. M. Shub. Process-originated migration in a heterogeneous environment. In *ACM Annual Computer Science Conference*, Feb. 1989.

[9] S. Dutta, R. Jensen, and A. Rieckmann. Viper: A multiprocessor SoC for advanced set-top box and digital TV systems. *Design & Test of Computers, IEEE*, 18(5), 2001.

[10] R. Fernandes, K. Pingali, and P. Stodghill. Mobile MPI programs in computational grids. In *Symposium on Principles and Practice of Parallel Programming*, Mar. 2006.

[11] A. Ferrari, S. J. Chapin, and A. Grimshaw. Heterogeneous process state capture and recovery through process introspection. *Cluster Computing*, 3(2), 2000.

[12] M. Hill and M. Marty. Amdahl's law in the multicore era. *Computer*, July 2008.

[13] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the cell multiprocessor. *IBM Journal of Research and Development*, July 2005.

[14] F. Karablieh, R. Bazzi, and M. Hicks. Compiler-assisted heterogeneous checkpointing. Oct. 2001.

[15] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen. Single-ISA Heterogeneous Multi-core Architectures: The Potential for Processor Power Reduction. In *International Symposium on Microarchitecture*, Dec. 2003.

[16] R. Kumar, D. M. Tullsen, N. Jouppi, and P. Ranganathan. Heterogeneous chip multiprocessors. *Computer*, 38(11), 2005.

[17] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas. Single-ISA Heterogeneous Multi-core Architectures for Multithreaded Workload Performance. In *International Symposium on Computer Architecture*, June 2004.

[18] MIPS Technologies, Inc. *MIPS32 Architecture for Programmers Volume II: The MIPS32 Instruction Set*.

[19] B. Ramkumar and V. Strumpen. Portable checkpointing for heterogeneous archtitectures. *International Symposium on Fault-Tolerant Computing*, June 1997.

[20] A. C. Ray and R. Hookway. DIGITAL FX!32 running 32-bit x86 applications on alpha NT. In *USENIX Windows NT Workshop*, Aug. 1997.

[21] C. M. Shub. Native code process-originated migration in a heterogeneous environment. In *ACM Conference on Cooperation*, Feb. 1990.

[22] J. Smith and R. Nair. *Virtual Machines: Versatile Platforms for Systems and Processes*. Morgan Kaufmann Publishers Inc., June 2005.

[23] P. Smith and N. C. Hutchinson. Heterogeneous process migration: the Tui system. *Software — Practice and Experience*, May 1998.

[24] B. Steensgaard and E. Jul. Object and native code thread mobility among heterogeneous computers (includes sources). In *Symposium on Operating Systems Principles*, Dec. 1995.

[25] V. Strumpen. Compiler technology for portable checkpoints. Technical report, Laboratory for Computer Science, Massachusetts Institute of Technology, 1998.

[26] V. Strumpen and B. Ramkumar. Portable checkpointing and recovery in heterogeneous environments. Technical report, University of Iowa, June 1996.

[27] Texas Instruments Inc. *OMAP5912 Multimedia Processor Device Overview and Architecture Reference Guide*.

[28] R. Veldema and M. Philippsen. Near overhead-free heterogeneous thread-migration. In *Cluster Computing*, Sept. 2005.

[29] D. G. von Bank, C. M. Shub, and R. W. Sebesta. A unified model of pointwise equivalence of procedural computations. *ACM Transactions on Programming Languages and Systems*, 16(6), 1994.

[30] C. Zheng and C. Thompson. PA-RISC to IA-64: Transparent execution, no recompilation. *Computer*, Mar. 2000.