Fast Dynamic Binary Rewriting for Flexible Thread Migration on Shared-ISA Heterogeneous MPSoCs

Georgakoudis, G., Nikolopoulos, D., Vandierendonck, H., & Lalis, S. (2014). Fast Dynamic Binary Rewriting for Flexible Thread Migration on Shared-ISA Heterogeneous MPSoCs. In 2014 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV). (pp. 156). Institute of Electrical and Electronics Engineers (IEEE). DOI: 10.1109/SAMOS.2014.6893207

Published in:
2014 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV)

Document Version:
Peer reviewed version

Fast Dynamic Binary Rewriting for Flexible Thread Migration on Shared-ISA Heterogeneous MPSoCs

Giorgis Georgakoudis
Queen's University of Belfast and University of Thessaly
g.georgakoudis@qub.ac.uk, ggeorgak@inf.uth.gr

Dimitrios S. Nikolopoulos
Queen's University of Belfast
d.nikolopoulos@qub.ac.uk

Hans Vandierendonck
Queen's University of Belfast
h.vandierendonck@qub.ac.uk

Spyros Lalis
University of Thessaly
lalis@inf.uth.gr

***Abstract*—Heterogeneous MPSoCs where different types of cores share a baseline ISA but implement different operational accelerators combine programmability with flexible customization. They hold promise for high performance under power and area limitations. However, transparent binary execution and dynamic scheduling is hard on those platforms. The state-of-the-art approach for transparent accelerated execution is fault-and-migrate (FAM): when a thread executes an accelerating instruction unavailable on the host core, it is *forcibly* migrated to an accelerating core which implements the instruction natively. Unfortunately, this approach prohibits dynamic scheduling through flexible thread migration, which is essential to any asymmetric platform for efficient utilization of heterogeneous resources.**

We present two distinct binary-level techniques – Dynamic Binary Rewriting (DBR) and Dynamic Binary Translation (DBT) – which enable selective acceleration, while preserving transparent thread execution and migration, to any core in the system, at any point in time. DBR rewrites binary code to exploit any accelerating instructions available in the host core. DBT implements a *fault-and-rewrite* scheme, which sets up trampolines to emulation routines for these accelerating instructions which are not available on the host core. Both methods customize binary code on demand, enabling flexible migration.

We evaluate the overhead of DBR and DBT against FAM on a real hardware shared-ISA MPSoC prototype. Experiments with single-thread programs show flexible migration is possible with manageable overhead. We measure the performance of our binary-level techniques by artificially triggering periodic thread migration between a Base and an accelerating (ACC) core. Periodic migration, without aiming for optimized scheduling, results in an average slowdown of about 40% under DBR or about 10% under DBT, compared to FAM driven scheduling. We also show results for a speedup proportional dynamic scheduler, enabled by our techniques, using multi-program workloads. In this case, up to 50% faster execution times can be achieved by leveraging flexible thread migration.

I. Introduction

Asymmetric multi-core architectures have shown to provide higher performance and better power optimization opportunities than symmetric ones. Asymmetric disjoint-ISA systems, such as the STHORM platform [1] and ARM Tegra family MPSoCs, have high performance potential but are hard to program, as they require intimate understanding of the functional heterogeneity between cores. On the other hand, asymmetric single-ISA architectures, such as ARM's BigLittle platforms [2] have performance asymmetry that is completely transparent to software. They are easier to program than disjoint-ISA multi-cores due to full binary compatibility between cores. However, single-ISA multi-cores offer no options for functional acceleration to applications that can leverage such acceleration for their critical kernels.

Recently proposed shared-ISA architectures [3], [4] bridge the dichotomy between disjoint-ISA and single-ISA multi-cores. On shared-ISA architectures all cores provide a common, baseline ISA, which preserves binary compatibility and improves programmability. At the same time, selected cores provide functional specialization, which is visible to software through ISA extensions. Software can exploit the acceleration potential of accelerating (ACC) cores, through code specialization. However, this instruction-based asymmetry renders execution non-transparent at the binary level. Binaries must forgo acceleration and implement only the baseline ISA to execute on all cores for full system utilization. If binaries include accelerating instructions they can execute only on ACC cores.

The state-of-the-art technique for transparent execution on shared-ISA systems is *fault-and-migrate* (FAM) [4], [5]. In this case, binaries include accelerating instructions but a thread can execute in any core in the system. If a thread executes an accelerating instruction on a core lacking it, FAM forcibly migrates the thread to an ACC core which implements this instruction. To alleviate congestion on ACC cores, FAM moves back a thread to its originator core after a *migrate-back* timeout expires. This approach has severe limitations: (i) FAM forced migration sets dynamically core affinities on a per-thread basis, precluding any other dynamic scheduling from implementing global optimization policies, (ii) it may over-subscribe ACC cores with threads and limit acceleration of those threads due to time-sharing the ACC core, (iii) FAM may suffer from an excessive number of unavoidable thread migrations and the associated loss of performance. Most importantly, FAM forced migration prohibits any dynamic scheduling approach to improve performance or power consumption through efficient utilization of heterogeneous resources.

[1]The research presented in this paper has received funding from the UK Engineering and Physical Sciences Research Council (grants EP/L004232/1, EP/L000055/1, EP/K017594/1) and the European Commission's Seventh Framework Programme (grants FP7-610509, FP7-323872). Hans Vandierendonck is supported by the People Programme (Marie Curie Actions) of the European Union's Seventh Framework Programme (FP7/2007-2013) under REA grant agreement no. 327744 (NovoSoft)

In this paper we propose two different techniques to support transparent accelerated execution and enable flexible, cross-core migration in shared-ISA systems. Through our techniques, a binary can execute on any core in the system, using accelerating instructions when possible. Also, any thread can migrate to any core type, at any point in time allowing dynamic schedulers to leverage flexible migration for implementing global optimizing policies. Specifically our contributions are:

- A lightweight Dynamic Binary Rewriting (DBR) method which rewrites binary code implemented in the baseline ISA, depending on the host core accelerating instruction extensions. DBR discovers code execution paths at runtime and replaces baseline instructions with accelerating instructions when executing on an ACC core or reverses previous code changes to execute on a Base core.
- A fast Dynamic Binary Translation (DBT) technique which implements *fault-and-rewrite*: when an accelerating instruction execution faults, DBT sets up a trampoline to an emulation routine translating this instruction to the baseline ISA. DBT reverts trampoline jumps to the original accelerating instruction for native execution on an ACC core.
- We evaluate our techniques against FAM on a hardware prototype of a shared-ISA MPSoC using single-program workloads from the SPEC CPU2006 [6] and Rodinia [7] suites. We measure the performance of DBR and DBT by triggering migration periodically between a Base and an ACC core. Periodic migration under DBR has an average slowdown of about 40% while DBT average slowdown is around 10% compared to FAM driven scheduling. This slowdown can be readily recovered through informed dynamic scheduling instead of arbitrary periodic migrations.
- We show results for a *S*peedup *P*roportional *D*ynamic *S*cheduler (SPDS) leveraging flexible migration as enabled by our binary-level techniques. SPDS dynamically migrates threads between Base and ACC cores according to their speedup. We compare it against FAM scheduling by running speedup heterogeneous, multi-program workloads on a hexa-core MPSoC consisting of 2 ACC and 4 Base cores. SPDS, employing our binary-level techniques, out-performs FAM scheduling by as much as 50%.

The rest of this paper is organized as follows: Section 2 briefly discusses fault-and-migrate and other transparent executions schemes. Section 3 presents our dynamic binary rewriting method and section 4 describes dynamic binary translation. Section 5 presents the evaluation of our work, including the experimental methodology and implementation details. Section 6 reviews related work. Section 7 concludes the paper.

II. Background on transparent execution

DBR and DBT are alternatives to two known methods that also enable software transparent execution on shared-ISA, asymmetric multi-core processors: Fault-and-Migrate [4], [5], [8] and universal binaries. For the sake of simplicity and without loss of generality, we will assume systems with two types of cores, Base and ACC, for the rest of the discussion.

The fundamental difference of our binary-level techniques compared to these methods is that they enable flexible migration which can be leveraged by dynamic schedulers to implement global optimizing policies. By contrast, FAM enforces thread migrations by the necessity to execute accelerating instructions only on ACC cores. FAM implicitly assumes that forced migration will opportunistically accelerate execution to offset the migration cost. However, this approach disrupts load balancing by oversubscribing ACC cores and hinders any global dynamic scheduling due to forcing core affinities. Universal binaries on the other hand forbid thread migration and enforce static, application-level scheduling decisions that may compromise system performance during the execution of multi-program workloads.

Universal (also known as "fat") binaries bundle together different versions of binary images for every ISA available on an asymmetric multi-core. At deployment time, the host assigned to execute the binary selects a compatible image for execution. However, universal binaries are designed for static allocation of binary images to cores with a matching ISA. They lack inherent support for thread migration: code deployed on a specific ISA must execute only on cores with the same ISA. Cross-architectural migration between distinct ISAs requires dynamic transformation of the binary code itself or the runtime state, such as the call stack. Such migrations, however, can be too costly or even infeasible, if there is not enough state information to resume a thread on a core with a different ISA [9].

FAM dynamically enables transparent binary execution at the cost of forced migrations. Under a FAM execution regime, code is compiled to target the full set of ACC instructions, as if the binary will always execute on an ACC core. However, a thread can still be placed to any core in the system. During execution, if a thread causes an illegal instruction exception on a Base core, it is forcibly migrated to an ACC core. Because of the necessity of FAM migrations, ACC cores may become over-subscribed and system performance may suffer from load imbalance. Furthermore, a scheduler that uses FAM has no option to select which thread(s) to accelerate using ACC instructions, when such a choice would affect application or overall system performance.

With our binary-level techniques, a binary can execute transparently to any core in the system, without forcing migration or any other disruption in scheduling. Also DBR and DBT differ from other frameworks that aim at binary portability, including interpreted execution, virtual machines and heavyweight binary rewriters, in that they do not necessitate continuous execution monitoring, costly binary analysis or prior code instrumentation. In particular, DBR operates on stripped binaries, and discovers live execution paths and rewriting opportunities for accelerating instructions only once, storing metadata to enable rewriting on demand. Likewise, DBT needs no binary instrumentation and incurs significantly lower overhead because it is invoked on demand and rewrites only in the scope of a single faulted instruction.

III. Dynamic Binary Rewriting

A. Overview

Dynamic Binary Rewriting (DBR) requires code to be compiled statically for targeting baseline instructions, to be specialized later with accelerating instructions depending on

the host core ISA. DBR is an OS service aware of the host core ISA type, while being transparent to application-level software. The OS bootstraps DBR on thread creation to do binary analysis and perform code specialization if the host core implements the ACC ISA. DBR is invoked again on thread migration to specialize code for the target core ISA.

DBR dynamically discovers execution flow paths by instrumenting branch instructions. It identifies rewriting targets on discovered basic blocks using peephole analysis. Depending on the host core ACC ISA, DBR patches binary code, replacing baseline instructions with accelerating instructions. In case a thread migrates to a Base core, the OS invokes DBR to revert previously rewritten code to the original baseline instruction implementation. Hence, unrestricted cross-core migration is possible at any point in time, with accelerated execution when the host core implements the ACC ISA. DBR operates on stripped binaries without prior instrumentation. It is designed to be lightweight by minimizing time spent during managed execution and aiming for native execution whenever possible.

B. Dynamic control flow discovery

The OS bootstraps DBR control flow analysis by providing the thread's start function, passed on from a *pthread_create* call. DBR dynamically discovers the execution path by following control flow instructions and discovering basic blocks. Discovered basic block info is stored in a list, in the OS thread descriptor structure, to be checked later for avoiding analysis on already discovered blocks. DBR disassembles instructions to find the branch instruction exiting the block. The target address of a direct branch can be resolved statically during binary analysis. We discuss indirect branches later. Control flow analysis creates a new basic block datum for the target and inserts it in the thread list. The datum stores the block starting address, which is the branch target, and the original instruction at this address. DBR replaces this instruction with a *software break* to a DBR entry routine and resumes native execution. If the break does execute, the entry routine saves thread context to restore it upon resuming native execution and invokes DBR control flow analysis. The analysis routine replaces the break with the original instruction and repeats the basic block discovery process. Branch targets within already discovered basic blocks are not followed, since code has been already examined. By setting break instructions on target addresses, DBR discovers only live code to save analysis overhead. For example this applies to conditional branches which may or may not be taken.

Indirect branches Although the target address of direct branches can be calculated statically, indirect branch targets can be only resolved at runtime. When control flow analysis encounters an indirect branch instruction, DBR replaces this instruction with a software break and inserts a helper basic block datum to save the original instruction. When the indirect branch software break executes, DBR resolves the branch target by consulting the saved thread context. DBR proceeds as with direct branch targets, inserting another basic block datum for the target and replacing the target address instruction with a software break. Indirect branch analysis is a significant source of overhead for traditional binary instrumentation frameworks [10]. Each execution of an indirect branch would need to break into DBR. However, indirect branches have good target locality [11] and most of the time DBR resolves the same target for consecutive executions. DBR mitigates indirect branch analysis overhead by following a *sampling approach* for processing indirect branches. The original indirect branch instruction is restored to replace the software break instruction, immediately after the branch target is resolved. At each scheduling interval, the OS invokes DBR to reset indirect branch instructions to software breaks for re-probing the target address if needed. In other words, indirect branches are sampled once every scheduling interval. This sampling strategy allows hotspot code targeted by indirect branches to be discovered without the overhead of continuous analysis.

```
1 ...
2 imm 0xc000
3 brlid r15, divsi3
4 addik r5, r5, 6
5 ...
```

(a) Original code

```
...
nop
addik r5, r5, 6
idiv r3, r6, r5
...
```

(b) Patching an accelerating integer division instruction

Fig. 1. Rewriting an accelerating instruction instead of a software emulation routine

C. Binary rewriting

At the same time with execution path discovery, DBR identifies rewriting targets through peephole analysis. Rewriting targets are saved in a list within the OS thread descriptor for future reference. The rewriting list is updated as new basic blocks are discovered. In the current implementation, DBR identifies calls to routines emulating unavailable ACC instructions with baseline instructions. If the host core implements the ACC ISA, DBR patches the binary code replacing the routine call with the actual accelerating instruction. DBR patching follows the routine call ABI to correctly setup the register operands for the accelerating instruction.

The OS invokes DBR before a thread migrates to a different core type in order to apply or undo patches on identified rewriting targets. If a previously rewritten thread running on an ACC core is migrated to a Base core, DBR reads the rewriting list and undoes changes in the binary code. Conversely, if a thread migrates from a Base core to an ACC core, DBR patches accelerating instructions for the discovered rewriting targets.

Rewriting happens in-place, on the application binary code itself. DBR may need to remove, reorder instructions, or amend the machine state because of rewriting. Figure 1 shows an indicative example. A routine call implementing integer division can be replaced with an accelerating *idiv* instruction in the ACC ISA. The call is implemented as a delay-slotted, branch-and-link instruction, preceded by an *imm* instruction for extending the branch operand address. DBR writes a *nop* in place of the *imm* instruction. Furthermore, it swaps places between the patched, accelerating *idiv* instruction and the instruction in the delay slot to preserve execution order.

DBR may need to update the thread's machine state due to rewriting. For example, assume a thread executes the rewritten code snippet in Figure 1(b) on an ACC core. If the thread is set to migrate to a Base core, DBR will undo the patch and inspect the *context-switch resume register* (CSRR). The CSRR

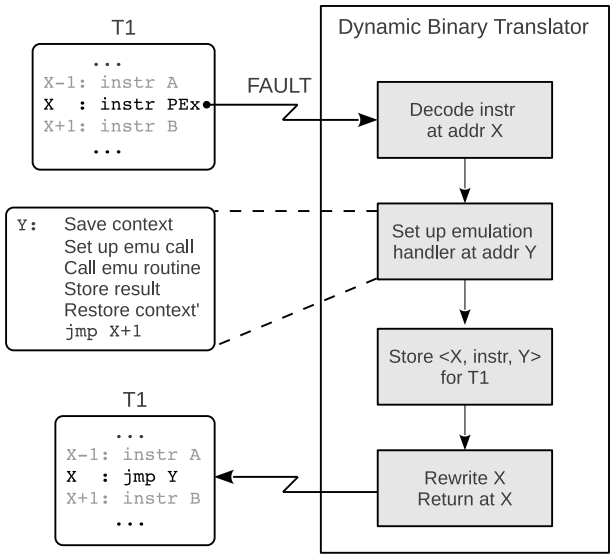

Fig. 2. An PE instruction fault triggers DBT which sets up a trampoline to an emulation routine

points to the application code address saved before entering the OS. This is the address the thread will resume execution after migration. If CSRR points to the instruction at line (3), DBR will move CSRR to the preceding instruction at line (2) to correctly resume at the restored routine call. In a more challenging case, if CSRR points to the previously patched accelerating instruction at line (4), the originally delay-slotted instruction has been already executed. Thus DBR cannot backtrack CSRR in the original code for correct execution after migration. In this case, DBR executes the accelerating instruction in its own context, updates the thread machine context as if the instruction has executed normally. CSRR is set to point to the subsequent instruction at which the application thread will resume after migration. DBR detects such cases and performs any necessary amendments.

IV. Dynamic Binary Translation

Dynamic Binary Translation (DBT) enables transparent execution and flexible migration by assuming binary code has been statically compiled for the ACC ISA. DBT is implemented as an OS service too, being transparent to the application layer. DBT operates by replacing ACC instructions with equivalent, lightweight emulation routines, or vice versa. Rewriting is triggered either when a thread executes an ACC instruction on a Base core causing a fault, or by a scheduling decision to migrate a thread from a Base core to an ACC core, for policy reasons. DBT replaces the faulted ACC instruction on a Base core with a trampoline to an equivalent emulation routine implemented with baseline instructions. Conversely, DBT restores trampolines to the original ACC instructions when a thread migrates from a Base core to an ACC core which is able to execute ACC instructions natively.

A. DBT operation

Figure 2 depicts DBT operation triggered by an instruction fault. In more detail, ACC instructions execute at full speed, without DBT intervening, when they are implemented in hardware. If a thread executes an ACC instruction on a Base core, an illegal instruction exception triggers the OS exception handler which invokes the DBT management routine. Firstly, DBT sets up an *emulation handler* for the faulted ACC instruction, translating it to an equivalent implementation in the baseline ISA. Secondly, it patches the faulting address with a trampoline jump to the emulation handler. Specifically, the emulation handler code performs the following sequence of operations:

(1) saves part of the execution context to avoid altering it when emulating
(2) sets up the call to the emulation routine, loading registers with input
(3) calls the emulation routine which is functionally equivalent to the ACC instruction
(4) stores back emulation results, saving them to output registers
(5) restores the unmodified execution context and finally,
(6) jumps back to the instruction following the trampoline branch.

Finally, DBT gives control back to the application to resume at the faulted address where the trampoline has been written. Execution resumes natively and jumps for the first time to the emulation handler code. Subsequent executions of the rewritten instruction address take the trampoline jump, without DBT intervening.

DBT stores info about rewriting, that is a data triplet containing the faulted address, the binary encoding of the faulted instruction and the emulation handler's address. This triplet is inserted in a per-thread rewriting list, saved within the OS thread descriptor. If a previously DBT-rewritten thread migrates to an ACC core, the OS invokes DBT to revert any patches made. DBT reads the thread's rewriting list and replaces trampoline jumps with the original ACC instructions for native execution after migration.

TABLE I. Table showing PE extension instructions

Instruction	Function	Hardware unit	Clock cycles
mul	INT multiplication	Integer Multiplier	1
idiv	signed INT division	Integer Divider	32
idivu	unsigned INT division	Integer Divider	32
fadd	FP addition	FPU	4
frsub	FP subtraction	FPU	4
fmul	FP multiplication	FPU	4
fdiv	FP division	FPU	4
fcmp	FP comparison	FPU	4

V. Evaluation

A. HW platform and SW support

We implemented a shared-ISA multi-core architecture on real hardware, an FPGA prototype that consists of ISA asymmetric Microblaze cores. This platform choice is motivated by the FPGA's extensibility for architectural exploration. Microblaze implements a 32-bit RISC ISA which can be extended with accelerating instructions, by optionally implementing additional hardware units. In our platform, ISA asymmetry comes

from enabling hardware implementation of computation accelerating components, i.e., the FPU and fast integer multiplier/-divider units, similar to previously proposed approaches [4]. Table I lists the respective accelerating instructions which can be offered as an extension of the baseline ISA.

Other than ISA based asymmetry, cores have the same micro-architectural characteristics: 100 MHz clock frequency, single issue in-order 5-stage pipeline, separate 32KB L1 instruction and data caches, a common, unified 512KB L2 cache, a 512-entry branch target cache, and a dedicated connection to the external memory controller.

The platform runs the Xilkernel OS with extensions that we introduced for multi-core support. The platform has no hardware support for cache coherence and cache consistency is implemented only for shared OS data, through software-controlled cache invalidations.

B. Implementation details

We extend the OS to implement DBR and DBT as system-level services transparent to the application level. The OS bootstraps DBR's code discovery with the thread's start routine for initializing dynamic binary analysis and instruction patching. On thread migration to a different ISA core, the OS invokes DBR to specialize code for the target core. DBR patches accelerating instructions if the target core implements the ACC ISA, or reverts previous patches to baseline instructions if the target is a Base core. DBR stores a total of 36 bytes of metadata per rewriting target, including the target's rewriting type, the target instruction address, the instruction itself, a flag for delay-slotted execution and other data structure variables. Also, DBR stores 68 bytes per basic-block discovered, including the instruction at the block start address, the block's start and end addresses and other data for fast searching in the basic block list.

Regarding DBT, the OS invokes it when a thread triggers an illegal instruction fault by executing an unavailable accelerating instruction on a Base core. Also, DBT is invoked in case a thread migrates from a Base core to an ACC core, to rewrite any patched trampoline jumps to the original accelerating instructions. DBT stores emulation handlers in core-private, low-latency scratchpad memories which are 32KB in size for our implementation. Scratchpads function as emulation handling buffers with fast, cache-like access times (1 cycle) and are addressable with a single branch instruction from the trampoline. DBT saves rewriting management data in per-thread OS descriptors for later reference, when managing the handler buffer or reverting back rewritten instructions. Specifically, DBT stores a total of 28 bytes per rewriting target including the target instruction's address, the instruction itself, and the emulation handler's address. On migration from a Base to an ACC core, DBT flushes any thread emulation handlers resident in the local scratchpad buffer and restores previously emulated ACC instructions to execute natively at the target core.

C. Experimental methodology

For the evaluation we use single-program or multi-program workloads where each program runs in single-threaded mode. Specifically, we port benchmarks from the SPEC CPU2006 [6] and Rodinia [7] suites to our platform. Table II lists the ported benchmarks. The table shows the executed instruction breakdown per-benchmark and the *end-to-end speedup* when the benchmark executes alone on a Base core using only baseline instructions versus executing on an ACC core using ACC instructions.

We categorize benchmarks based on their speedup as *High-speedup*, *Medium-speedup* and *Low-speedup*. High-speedup benchmarks achieve more than $10\times$ acceleration from ACC instructions. This class includes *streamcluster*, *cfd*, and *lud*, which are computational kernels from the Rodinia suite making frequent use of FP operations. Medium-speedup benchmarks achieve speedup between $10\times$ and $2\times$. These are: *milc*, *bfs*, *namd*, *srad*, *backprop*, *hmmer*. Medium-speedup benchmarks have most their speedup achieved due to accelerating integer instructions and to a lesser extend from infrequent use of FP operations. Interestingly, compiling *milc* and *namd*, which are part of SPEC FP benchmarks, produces a binary which does not use hardware, single-precision FPU instructions because of double-precision FP arithmetic in the code. Nevertheless, hardware integer instructions accelerate double-precision compiled code, providing more than $3\times$ speedup over baseline instructions. Benchmarks that achieve less than $2\times$ speedup from ACC instructions are categorized as low-speedup ones and include: *sjeng*, *h264ref*, *hotspot*, *astar*, *libquantum* and *bzip2*. Those benchmarks execute mostly baseline instructions and very few accelerating ones.

For evaluating the overhead of our binary-level techniques compared to FAM we use a platform configuration with one ACC core and one Base core. Each benchmark runs alone in single-threaded mode. When evaluating FAM the benchmark thread is initially placed on the Base core. FAM induces forced migration to the ACC core based on instruction faults while the OS moves back the thread to the Base core when the migrate-back timeout expires. We perform experiments with various values for the migrate-back timeout.

DBR and DBT enable flexible migration at any point in time, hence ideally, migration decisions should be driven by an optimizing dynamic scheduler. However, we experiment with a periodic migration approach to help quantify overhead and show an informed dynamic scheduler later. Specifically, the OS artificially triggers migration between the Base and the ACC core, or vice versa, at periodic intervals, and invokes DBR or DBT as needed.

We measure the benchmark *turnaround time* in quantum scheduling intervals for both FAM and our binary-level techniques. The migrate-back timeout for FAM and the migration period for DBR and DBT range from 1 to 16 scheduling intervals. Note that in our implementation, the quantum scheduling interval is 10ms.

D. Results

We discuss the performance results of induced periodic migration under our binary-level techniques and FAM opportunistic acceleration. Figure 3 shows the turnaround time of each benchmark executing under FAM, DBR or DBT for various migration periods. Periodic migration under DBR has an average slowdown of around 40% across all benchmarks and migration periods while DBT slowdown is less, about 10%,

TABLE II. BENCHMARKS FROM SPEC CPU2006 AND RODINIA SUITES

Benchmark	Executed instructions breakdown (% of total)									$SP = \frac{TT_{Base}}{TT_{ACC}}$	SP class
	mul	idiv	idivu	fadd	frsub	fmul	fdiv	fcmp	base		
streamcluster	0.11	-	-	4.33	4.46	4.47	-	0.13	86.50	24.27	High
cfd	0.29	-	-	4.81	0.66	5.34	0.59	0.40	87.91	14.4	High
lud	3.79	-	-	-	3.78	3.78	0.01	-	88.64	14.25	High
milc	3.53	-	-	-	-	-	-	-	96.47	4.7	Medium
bfs	2.25	-	0.53	-	-	-	-	-	97.22	3.9	Medium
namd	3.06	-	-	-	-	-	-	-	96.94	3.59	Medium
srad	1.90	-	-	0.09	0.05	0.12	0.04	0.02	97.78	3.4	Medium
backprop	0.55	-	-	0.16	-	0.17	-	-	99.12	2.63	Medium
hmmer	0.62	0.04	-	0.20	-	-	-	0.20	98.94	2.08	Medium
sjeng	0.68	-	0.05	-	-	-	-	-	99.27	1.53	Low
h264ref	0.61	0.04	0.02	-	-	-	-	-	99.33	1.37	Low
hotspot	0.15	-	-	-	-	-	-	-	99.85	1.14	Low
astar	0.08	-	-	0.02	-	0.03	-	-	99.87	1.13	Low
libquantum	0.08	-	-	0.01	-	0.02	0.01	-	99.88	1.05	Low
bzip2	0.01	-	-	-	-	-	-	-	99.99	1.01	Low

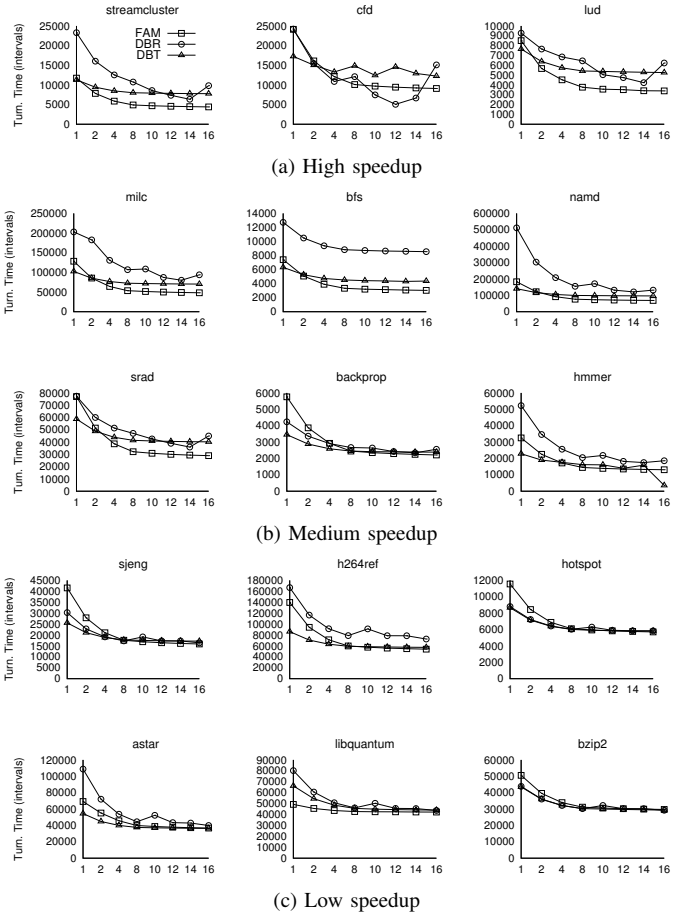

Fig. 3. Comparing performance of DBR, DBT periodic migration vs. FAM. The horizontal axis denotes migration periods for DBR, DBT and migration-back timeout values for FAM.

compared to FAM. The reason for DBR greater slowdown is that it operates on code statically targeted for the baseline instructions. Statically targeted code for the ACC ISA, used by FAM (and DBT), may emit additional accelerating instructions during code generation which might not be identifiable in baseline code through fast peephole analysis. Furthermore, the compiler statically targeting the ACC ISA may do extra, ISA-neutral, compile-time code optimizations exposed by code generation for the ACC ISA. Nevertheless, DBR can be extended with more sophisticated binary analysis which we leave as future work.

Moreover, FAM opportunistically migrates threads to ACC cores on an instruction fault for accelerated execution. In single-program workloads there is no time sharing of ACC cores and a thread is accelerated fully with only the penalty of instruction fault handling and migration, mitigated when the migrate-back period increases. On the other hand, our periodic migration approach does not perform any kind of scheduling optimization. For example, under FAM, High-speedup benchmarks execute mostly on the ACC core due to frequent forced migration from instruction faulting on the Base core. However, under induced periodic migration, a benchmark thread will execute a full period on the Base core before it is migrated to the ACC core and vice versa.

These results help quantify the cost of DBR and DBT using a periodic migration scheme. Notably, the flexibility for cross-core execution enabled by our binary-level techniques permits a dynamic scheduler to implement global optimizing policies instead. We present results for an informed dynamic scheduler leveraging flexible migration next.

E. Informed dynamic scheduling

We show results for a dynamic scheduler to illustrate the efficiency of flexible migration enabled by our binary-level techniques. The dynamic scheduler implements a *speedup proportional policy* for time sharing ACC cores between executing threads in proportion to each thread's speedup.

We briefly discuss the *S*peedup *P*roportional *D*ynamic *S*cheduler (SPDS) internals. The scheduler operates in *rounds* for time sharing ACC cores. A round is a system-wide scheduling epoch which consists of a fixed number of quantum scheduling intervals. During a round, the scheduler monitors the number of the number of intervals each thread has executed on an ACC core. It employs a thread swapping mechanism, migrating threads between Base and ACC cores, so that each thread executes on an ACC core for a number of intervals which is proportional to its speedup, compared with the speedup of the other threads in the system. At the end of a round, the scheduler resets the ACC interval counters of all threads to begin a new round. For our implementation we retrofit each benchmark's speedup to the scheduler, as shown in table II, and set the round duration to 100 quantum scheduling intervals.

TABLE III. SPEEDUP HETEROGENEOUS, MULTI-PROGRAM WORKLOADS

	Workload
3H-3L	cfd, streamcluster, lud, sjeng, bzip2, hotspot
3H-3M	cfd, lud, streamcluster, milc, backprop, hmmer
3M-3L	hmmer, srad, backprop, bzip2, sjeng, hotspot

For the evaluation we configure our MPSoC platform so that it includes 2 ACC and 4 Base cores. In our experiments the system is kept fully-subscribed running speedup heterogeneous, multi-program workloads. This means that multi-program workloads contain as many benchmarks as the total number of cores (irrespective of core type) and that after a benchmark completes execution, it is restarted to ensure a fully-subscribed system throughout workload execution. An experiment is deemed finished when each benchmark in the workload has completed at least three runs, i.e., the slowest thread has completed its benchmark for the 3rd time and other threads are stopped having completed at least three runs of their respective benchmark. Table III shows the benchmarks used in each workload. A workload is denoted by the number of benchmarks from each speedup class (High, Medium, Low). For example, the workload denoted as 3H-3M has 3 High-speedup and 3 Medium-speedup benchmarks. We compute the *average turnaround time* for each benchmark by taking the mean of turnaround times of completed runs. Per-benchmark average turnaround times are aggregated to compute SPDS speedup over FAM scheduling to define *workload speedup*. The workload speedup is the geometric mean of per-benchmark speedup values, which are calculated as the ratio of the benchmark turnaround time executing under FAM over SPDS. Formally, workload speedup (WSP) is:

$$WSP_{SPDS/FAM} = \sqrt[N]{\prod_{b \in W} \frac{AvgTT_{b,FAM}}{AvgTT_{b,SPDS}}}$$

where N is the total number of benchmarks (6 in our setup) and the set $b \in W$ includes each benchmark in the workload.

For FAM scheduling, the (a priori known) highest speedup benchmarks threads known are placed on ACC cores at loading time to reduce the number of faults causing forced migrations. The FAM migrate-back timeout is set to 10 scheduling intervals which shows good performance based on the single-program evaluation. Initial thread placement for SPDS, either based on DBR (SPDS-DBR) or DBT (SPDS-DBT), is not important since dynamic scheduling will migrate threads for sharing ACC cores. In our implementation, SPDS-DBR and SPDS-DBT perform round balancing, that is thread swaps, every 10 scheduling intervals to be comparable with FAM's migrate-back timeout.

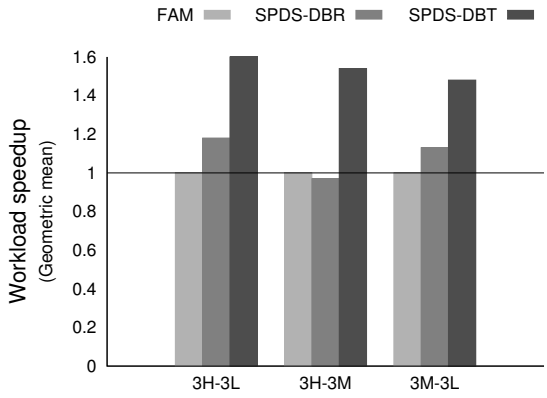

Fig. 4. SPDS-DBR and SPDS-DBT vs. FAM for multi-program workloads

Figure 4 shows the results. SPDS-DBT performs significantly better than FAM forced migration scheduling. It improves performance by about 50% compared to FAM, across all multi-program workloads. SPDS-DBR improves workload speedup in comparison to FAM by about 20% and 15% for the 3H-3L and 3M-3L workloads respectively, while it performs on par with FAM for the 3H-3M workload. This is because DBR works with code statically targeted for baseline instructions as we discussed in the previous section.

The results show that dynamic scheduling, enabled by flexible migration, can efficiently leverage our binary-level techniques for implementing optimizing policies to out-perform FAM forced migration scheduling.

VI. RELATED WORK

Shared-ISA heterogeneous architectures have been recently proposed [3], [4], [12]. Shared-ISA heterogeneous MPSoCs enable more flexible customization by breaking the single-ISA restriction through accelerating instruction extensions on certain cores in the system. There is previous work on dynamic scheduling to meet performance and power constraints on single-ISA heterogeneous MPSoCs [13], [14] assuming ISA transparent migration. Also, there are studies on static mapping techniques for disjoint-ISA systems [15], [16] targeting improved performance and power consumption.

Regarding shared-ISA MPSoC systems, Shen et. al. [17] propose several dynamic scheduling algorithms. However in their work, tasks are allowed to migrate only within a subset of cores that meet the task's ISA requirements. This approach imposes core affinities that limit flexibility. Our binary-level techniques allow thread migration to any core type in the system to unlock the full potential of dynamic scheduling despite software-visible instruction heterogeneity.

Fault-and-migration (FAM) [4], [18] has been proposed as a method for transparent execution with opportunistic acceleration on shared-ISA architectures. However, FAM forces

thread migrations which congest ACC cores and forbids the scheduler to implement global optimizing policies. We have extensively compared our work with FAM to show that flexible migration, provided by our binary-level techniques, enables dynamic scheduling which out-performs FAM's forced migration scheduling.

Discussing work on rewriting, existing dynamic binary rewriters, such as DynamoRIO [19] and Pin [20], execute a binary in managed mode. They need to manage software code caches to enable code profiling, security checking and complex optimizations. This approach permits elaborate code analysis and fine-grain binary instrumentation but comes with significant overhead. Instead, DBR and DBT are designed to be lightweight and fast, aiming for as much unobstructed execution as possible to be efficient. Also, traditional rewriters assume that code is immovable, which allows them to do time consuming optimizations, whereas our binary-level techniques target flexible, cross-core migration.

VII. Conclusions and Future Work

In this paper we presented two fast dynamic binary rewriting techniques, DBR and DBT, for flexible, cross-core thread migration in shared-ISA MPSoC platforms. DBR adapts binary code on demand for the host core ISA. DBT employs *fault-and-rewrite* in conjunction with a translation scheme to execute hardware unavailable accelerating instructions. Both methods achieve software transparent execution despite of ISA heterogeneity. Additionally, they enable flexible, cross-core thread migration in contrast to *fault-and-migrate* (FAM) which is the state-of-the-art approach for transparent execution on shared-ISA platforms. We have also shown a speedup-proportional dynamic scheduler which leverages flexible migration provided by our binary-level techniques to out-perform significantly FAM forced migration scheduling.

For future work we intent to expand our dynamic binary techniques to more classes of ACC instructions, such as SIMD extensions, which may require transformation on architectural state. We are also considering dynamic instruction rewriting for binary code adaptation affecting functional unit sharing and power consumption. Moreover, we work on dynamic scheduling algorithms, enabled by flexible migration, that aim to perform global performance and power optimization on shared-ISA asymmetric platforms.

References

[1] D. Melpignano, L. Benini, E. Flamand, B. Jego, T. Lepley, G. Haugou, F. Clermidy, and D. Dutoit, "Platform 2012, a many-core computing accelerator for embedded SoCs: Performance evaluation of visual analytics applications," in *Proceedings of the 49th Annual Design Automation Conference*, ser. DAC '12. New York, NY, USA: ACM, 2012, pp. 1137–1142. [Online]. Available: http://doi.acm.org/10.1145/2228360.2228568

[2] P. Greenhalgh, "Big. little processing with arm cortex-a15 & cortex-a7," *ARM White Paper*, 2011.

[3] E. Flamand, "Strategic directions towards multicore application specific computing," in *Design, Automation Test in Europe Conference Exhibition, 2009. DATE '09.*, April 2009, pp. 1266–1266.

[4] T. Li, P. Brett, R. Knauerhase, D. Koufaty, D. Reddy, and S. Hahn, "Operating system support for overlapping-isa heterogeneous multi-core architectures," in *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, 2010, pp. 1–12.

[5] D. Reddy, D. Koufaty, P. Brett, and S. Hahn, "Bridging functional heterogeneity in multicore architectures," *SIGOPS Oper. Syst. Rev.*, vol. 45, no. 1, pp. 21–33, Feb. 2011. [Online]. Available: http://doi.acm.org/10.1145/1945023.1945028

[6] J. L. Henning, "Spec cpu2006 benchmark descriptions," *SIGARCH Comput. Archit. News*, vol. 34, no. 4, pp. 1–17, Sep. 2006. [Online]. Available: http://doi.acm.org/10.1145/1186736.1186737

[7] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, 2009, pp. 44–54.

[8] J. C. Saez, A. Fedorova, D. Koufaty, and M. Prieto, "Leveraging core specialization via os scheduling to improve performance on asymmetric multicore systems," *ACM Trans. Comput. Syst.*, vol. 30, no. 2, pp. 6:1–6:38, Apr. 2012. [Online]. Available: http://doi.acm.org/10.1145/2166879.2166880

[9] M. DeVuyst, A. Venkat, and D. M. Tullsen, "Execution migration in a heterogeneous-isa chip multiprocessor," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVII. New York, NY, USA: ACM, 2012, pp. 261–272. [Online]. Available: http://doi.acm.org/10.1145/2150976.2151004

[10] J. D. Hiser, D. W. Williams, W. Hu, J. W. Davidson, J. Mars, and B. R. Childers, "Evaluating indirect branch handling mechanisms in software dynamic translation systems," *ACM Trans. Archit. Code Optim.*, vol. 8, pp. 9:1–9:28, June 2011. [Online]. Available: http://doi.acm.org/10.1145/1970386.1970390

[11] B. Dhanasekaran and K. Hazelwood, "Improving indirect branch translation in dynamic binary translators," in *Proceedings of the ASPLOS Workshop on Runtime Environments, Systems, Layering, and Virtualized Environments*, pp. 11–18.

[12] H. Shen and F. Ptrot, "Using Amdahls law for performance analysis of many-core SoC architectures based on functionally asymmetric processors," in *Architecture of Computing Systems - ARCS 2011*, ser. Lecture Notes in Computer Science, M. Berekovic, W. Fornaciari, U. Brinkschulte, and C. Silvano, Eds. Springer Berlin Heidelberg, 2011, vol. 6566, pp. 38–49.

[13] O. Arnold and G. Fettweis, "Power aware heterogeneous MPSoC with dynamic task scheduling and increased data locality for multiple applications," in *Embedded Computer Systems (SAMOS), 2010 International Conference on*, July 2010, pp. 110–117.

[14] C. Bechara, N. Ventroux, and D. Etiemble, "Comparison of different thread scheduling strategies for asymmetric chip multithreading architectures in embedded systems," in *Digital System Design (DSD), 2011 14th Euromicro Conference on*, Aug 2011, pp. 181–187.

[15] E. Carvalho and F. Moraes, "Congestion-aware task mapping in heterogeneous MPSoCs," in *System-on-Chip, 2008. SOC 2008. International Symposium on*, Nov 2008, pp. 1–4.

[16] J. Castrillon, R. Leupers, and G. Ascheid, "Maps: Mapping concurrent dataflow applications to heterogeneous MPSoCs," *Industrial Informatics, IEEE Transactions on*, vol. 9, no. 1, pp. 527–545, Feb 2013.

[17] H. Shen and F. Pétrot, "Novel task migration framework on configurable heterogeneous MPSoC platforms," in *Proceedings of the 2009 Asia and South Pacific Design Automation Conference*, ser. ASP-DAC '09. Piscataway, NJ, USA: IEEE Press, 2009, pp. 733–738. [Online]. Available: http://dl.acm.org/citation.cfm?id=1509633.1509798

[18] D. Reddy, D. A. Koufaty, P. Brett, and S. Hahn, "Bridging functional heterogeneity in multicore architectures," *Operating Systems Review*, vol. 45, no. 1, pp. 21–33, 2011.

[19] D. Bruening, T. Garnett, and S. Amarasinghe, "An infrastructure for adaptive dynamic optimization," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, ser. CGO '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 265–275. [Online]. Available: http://dl.acm.org/citation.cfm?id=776261.776290

[20] C.-K. Luk, R. S. Cohn, R. Muth, H. Patil, A. Klauser, P. G. Lowney, S. Wallace, V. J. Reddi, and K. M. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *PLDI*, 2005, pp. 190–200.