

Guest Editors' Introduction

Erik R. Altman, David Kaeli, Yaron Sheffer

The emergence of a new computer architecture is a rare event. Designing a new instruction-set architecture (ISA) is very expensive, and it is often necessary to recompile several operating systems and a large number of application programs. *Binary translation* attempts to alleviate much of this effort and the associated risk: recompilation is replaced by automatic conversion of code from the legacy ISA to the new one. Even with modern programming languages, language semantics are typically not defined tightly enough to make recompilation a transparent process. However, the semantics of binary code is usually well defined, facilitating automatic and transparent translation.

Binary translation has been practiced for many years. But only with the recent increases in processing power has it become possible to utilize translation fully.

In the last few years the field has expanded, including the development of a number of new binary translation systems and research projects. The field has matured significantly, evolving a set of common concepts.

We begin with a tutorial on binary translation, followed by three papers written by practitioners, each demonstrating a different approach:

- HP's dynamic translator, **Aries**, is a soon-to-be-released *dynamic translator*, designed to ease the transition of applications from HP's **Precision Architecture** to *IA-64*.
- **BOA** is an IBM research effort to translate legacy *PowerPC* code into a specially designed VLIW architecture. The paper discusses dynamic binary translation of a full system, rather than just application code.
- **UQBT** is a binary translation system from the University of Queensland. Instead of translating between a specific pair of instruction sets, this system enable binary translation to be generalized, working efficiently between virtually any two architectures.

The concept for this special issue originated in the *Workshop on Binary Translation* [5], which took place in October 1999 as part of the conference on **Parallel Architecture and Compilation Techniques (PACT)**.

The **BOA** and **UQBT** papers as well as several of the tutorial sidebars are based on the papers presented at the workshop.

We would like to take this opportunity to acknowledge the efforts of the workshop's program committee: *Brad Calder, Evelyn Duesterwald, Kemal Ebcioglu, Michael Franz, Michael K. Gschwind, Ray Hookway, David Keppel, Andreas Krall, James R. Larus, David J. Lilja, Soo-Mook Moon, Sumedh Sathaye, and Michael Smith*. We would also like to thank the organizers of PACT '99 for their support, in particular *Nader Bagherzadeh, Linda Llegh, Thomas Kim and Walid Najjar*.

Binary translation has been used to move from one computer architecture to another, to execute code whose source is long lost, to simulate new computer architectures, and to do performance analysis on programs for a given computer architecture.

A few of the notable developments related to binary translation include:

- In 1972, IBM shipped VM/370, a virtual machine system which supported System/360 or System/370 operating systems, such as DOS/360, OS/VS1, OS/VS2, CMS or any version of OS/360.
- In 1987, Hewlett-Packard shipped one of the earliest commercial binary translation systems, to migrate an existing *HP 3000* customer base from one computer architecture to another. Tandem Computers shipped an interpreter and translator system 1991.
- Starting in 1992, Digital Equipment Corporation shipped an ambitious series of binary translators to migrate users to the *Alpha* architecture. These included translators for *VAX/VMS*, *MIPS/Unix*, *SPARC/Unix*, and *x86/NT* to *Alpha*.
- Apple Computer shipped a *Motorola 68000* interpreter with the *PowerMac* in 1994, which was later upgraded to use translation. Binary translation has also been used to run Windows programs on a variety of computers.
- In 1996, Sun released the first version of the Java Development Kit.
- Also in 1996, the **DAISY** project was started at IBM Research.

Figure 1. SIDEBAR: Binary Translation History, — Richard L. Sites (Adobe Systems, Inc.)

Binary Translation: A Short Tutorial

1. Introduction

Binary translation is the automatic conversion of code from one instruction set to another. Binary translation involves an innovative mix of several disciplines, such as compiler technology, computer architecture, adaptive software, virtual machines and others. We discuss how binary translation benefits from each of them, and how they fit together to produce usable translation products.

When porting legacy code from an existing instruction set architecture (*legacy architecture*) to a new ISA (*target architecture*), a number of techniques can be used:

- Provide a special *processor mode* to execute legacy code on the new processor (e.g., the *PDP-11* mode on *VAX*). Such a processor mode may itself use binary translation behind the scenes. This will allow code images targeting the legacy and new architectures to run concurrently.
- Recompile the program to the new instruction set.
- Use a variety of software methods to interpret or translate the application. Such translation can be

done at run-time or off-line, and may benefit from analyzing the run time behavior of the program. This paper focuses on this last class of techniques.

Binary translation systems can be classified as:

- An **emulator (software emulator, interpreter)** interprets program instructions at run-time. Interpreted instructions are not saved or cached. Emulators are relatively easy to develop, and can be made highly compatible with the *legacy* architecture with modest effort.
- A **dynamic translator** translates pieces of code at run-time from the *legacy* to the *target* ISA, caching them for future use. *Java JIT (Just-In-Time)* compilers are probably the best-known translators in this class. Figure 2 describes **Latte**, which generates high quality *Sparc* code from *Java* bytecode.

A *dynamic translator* may be part of the translated code's execution thread. Hence, execution stalls will occur during translation. Alternatively, the translator may run asynchronously with the translated code. Dynamic translators often make use of program behavior information. The related

dynamic optimizer performs similar run-time optimization without the translation, i.e., transforming code within the same ISA. HP's *Dynamo* (see Figure 3), is a good example of a *dynamic optimizer*.

- A **static translator** translates programs in an off-line process and can apply more rigorous (i.e., time consuming) code optimizations than can a dynamic translator. Static translators can also use execution profiles obtained during a program's previous run.

All three approaches have limitations: both emulation and dynamic translation impose run-time overhead, while static translation as a stand-alone tool requires end-user involvement, which has impeded its widespread adoption. Much recent work in this area revolves around innovative *hybrid* solutions, to combine the best features of each approach. The Compaq FX!32 [8], which translates Windows NT applications compiled for *x86* to *Alpha* code, is such a solution, where an application is emulated if necessary, but the bulk of the optimization is done statically when the computer is idle. The next run of the program can use the highly-optimized translated code. Code sections that have not been translated are emulated.

2 Runtime Performance Optimization

For binary translation to serve as a viable alternative to *legacy* architecture execution, the performance of the translated *target* executable should be competitive with the *legacy* architecture performance. The *legacy* architecture executable has the luxury of being produced using an optimizing compilation process.

Binary translation in the general case does not have high-level language code available, and thus works solely from the executable code. Not knowing the full semantics of the original source code, a binary translator cannot perform many optimizations available to a compiler.

One common approach to improving binary translation performance is profile-guided optimization. As illustrated in Figure 4, execution profiles can be generated efficiently. These profiles are used to guide optimization performed during both the translation process, and, in a system like FX!32, during further tuning of the translated image (*target* optimization). Among the algorithms benefitting most from profile information are instruction scheduling and register allocation. A static translation system can merge multiple profiles

into a single profile. A dynamic translation system may retranslate translated code periodically as the profiles change.

Several other optimizations frequently used in binary translation are:

- **ISA remapping** - Handling register overlaps and code generation present in the *legacy* ISA and efficiently remapping them to the *target* ISA.
- **Basic block reordering** - Keeping the *target* image execution as sequential as possible, so that conditional branches will typically fall-through. This helps speed instruction fetch and improve instruction cache performance.
- **Memory coloring** - Improving the mapping of the translated image onto the memory hierarchy of the *target* environment [6].
- **Code specialization** - Cloning procedures based on the invariance of parameter values.

Any optimization considered by the binary translation system can be used only if the correctness of the translation is preserved! Correctness and performance are often competing issues, as discussed in [1].

3 Challenges of Legacy Code and Some Solutions

There are a variety of difficulties in translating from a legacy architecture such as *x86*, *PowerPC*, or *VAX* to a new machine such as *IA-64*, *BOA*, or *Alpha*. Since all machines — legacy and new — are Turing machines, any computation done on one can be emulated on another. As such, binary translation aspires to do more — to emulate the legacy architecture efficiently

(so efficiently that code for the legacy architecture runs as fast on the new architecture as on the legacy machine, and preferably faster). Attaining this goal requires careful design in a number of areas, some of which we touch upon here:

- The **instruction set** of the legacy architecture should be executed efficiently. This is typically a moving target, since architectures are never fixed. Sidebar 5 discusses the approach the FX!32 translator took in implementing the MMX instructions when they were added to the *x86* architecture.
- The **number of registers** in the new machine should be at least equal to the number in the

Programs written in *Java* are typically translated into bytecode, so that binary distributions of *Java* applications are portable to any platform for which a *Java* virtual machine (JVM) is available.

One approach to executing *Java* bytecode is to compile a set of bytecode instructions to native code “just-in-time” (JIT) to be executed. This can result in drastic improvements in performance compared to an interpreter. It is important that translation overhead be small, since it is part of the execution time.

LaTTe, available from <http://latte.snu.ac.kr/>, is one such JVM with a JIT compiler. The JIT compiler supports the *UltraSPARC*, and uses a new register allocation algorithm and also does other “traditional” and object-oriented optimizations.

Register allocation is very important, and is probably responsible for the largest performance improvement that a JIT compiler obtains compared to an interpreter. **LaTTe** uses a two-pass linear scan register allocation algorithm¹ on each extended basic block. In the first pass, a backward sweep constructs a preferred mapping of symbolic registers to actual registers, and in the second pass, a forward sweep constructs the actual mapping. This algorithm produces good results with low overhead.

The JIT compiler is not the only thing that determines the performance of a JVM. Thread synchronization, exception handling, and garbage collection are also important. If these are slow, then a JIT compiler can only do so much to improve performance. To wit, **LaTTe** uses a lightweight monitor for thread synchronization, on-demand translation of exception handlers, pointer incrementing memory allocation, and a fast partially conservative mark and sweep garbage collector. Consequently, the performance of the **LaTTe** JVM is competitive with that of commercial JVMs, such as SUN’s *Hotspot* and *JDK 1.2* production release.

LaTTe is a research prototype and has been used to test many optimizations, including fast register allocation, reduction of virtual call overheads, instruction scheduling, adaptive compilation, and new memory allocation and garbage collection algorithms. **LaTTe** has a liberal *BSD*-like license. We hope others will use it as a research framework. **LaTTe** was developed by the MASS Laboratory at Seoul National University with sponsorship from IBM Watson Research.

References

1. B.-S. Yang et al., “LaTTe: A Java VM Just-in-Time Compiler with Fast and Efficient Register Allocation,” *Proc. Int’l Conf. on Parallel Architectures and Compilation Techniques*, IEEE CS Press, Los Alamitos, Calif., 1999, pp. 128-138.

Figure 2. SIDEBAR: LaTTe: An Open-Source Java VM Just-in-Time Compiler, — Soo-Mook Moon (Seoul National University) and Kemal Ebcioğlu (IBM Research)

The conventional use of binary translation (BT) technology is in software migration across different instruction set architectures. *Dynamo*, a prototype dynamic optimizer developed at Hewlett-Packard Laboratories employs BT technology in a novel way, for the transparent acceleration of native program binaries. *Dynamo* operates entirely at runtime by dynamically generating optimized native re-translations of the running program's hot spots. Contrary to intuition, *Dynamo* demonstrates that it is possible to use a piece of software to improve the performance of a native, statically optimized program binary, while it is executing. For example, the performance of many SPECint95 binaries compiled with moderate optimization running under *Dynamo* is comparable to the performance of their highly statically optimized version running without *Dynamo*. *Dynamo*'s operation is transparent in the sense that it does not depend on any special programming language annotations, compiler generated hints, binary instrumentation, or OS/hardware support.

Dynamo uses lightweight profiling techniques to identify hot traces at runtime. The program behavior is monitored via (native instruction) interpretation, until the target of a backward taken branch becomes hot; at that point, the very next sequence of interpreted instructions is speculatively selected as a hot trace. The selected traces form a single-entry multi-exit region of code that is optimized using low-overhead optimization techniques and emitted into a software code cache. Traces may cross statically and dynamically linked procedure boundaries, potentially exposing optimization opportunities not available to a static compiler. Over time, as the program's working sets materialize in the software cache, execution takes place almost entirely inside the optimized code cache, thus giving a performance boost.

The *Dynamo* prototype is a realistic software implementation running on a *PA-RISC* machine under the *HPUX* operating system. *Dynamo* features a space-efficient design and a code size of less than 265 KB. Its compact memory footprint not only makes *Dynamo* deployable in a wide range of platforms, it also makes *Dynamo* a viable candidate for coupling with existing software systems. For example, *Dynamo* can be used as a performance backend for a dynamic binary translator. Further details are available at the *Dynamo* Project web site: <http://www.hpl.hp.com/cambridge/index.html>.

Figure 3. SIDEBAR: The Dynamo Dynamic Optimizer: Employing BT Technology for Native Binary Acceleration — Vasanth Bala (HP Labs), Evelyn Duesterwald (HP Labs), and Sanjeev Banerjia (InCert)

Efficient profiling is essential to runtime optimization. This has two facets:

1. Profiling overhead must be small so it can always be enabled, and thus allow programs to be continuously monitored for re-optimization. A typical runtime optimization system operates in two phases: (1) less-optimized execution with profile collection and (2) optimized execution without profile collection.
2. Profile information should be aggregated and ready for use in optimization. Sampling based techniques [1] could have low overhead, but they often require post-processing to derive profile information.

The approach in [2] achieves both objectives by combining two collaborative techniques: powerful compiler analysis that inserts minimal profiling instructions; and hardware that asynchronously executes profiling operations in free slots in user programs executed on wide-issue processors like *IA-64*. The compiler first analyzes the user program to determine the minimal set of blocks that need profiling. Next, it allocates profile counter space for each function and inserts an `iniprof` instruction in the function entry block to tell the hardware the starting address of the profile counter space. Then it inserts a `profile ID` instruction in each profiled block to pass profiling requests to the hardware. The hardware derives the profile counter address from the `iniprof` and `profile ID` instructions and generates *profile update* operations. These operations are executed when free slots exist during program execution.

Since most profiled blocks contain a branch instruction, we can eliminate most `profile ID` instructions by encoding the ID into 8 bits of a branch instruction. For large functions that require more `profile ID`'s than the 8-bit ID field can hold, we apply a graph partitioning algorithm to partition the control flow graph into single-entry regions such that the ID values in each region can be encoded into the 8-bit ID field. In this case, we insert a `setoffset` instruction in each region entry block to pass an offset value to the hardware.

The new profiling technique enables programs with profiling to run almost as fast as without profiling. For example, we observed less than 0.6% overhead for the `SPECint95` benchmarks.

[1] Jeff Dean, et al, “*ProfileMe: Hardware Support for Instruction-Level Profiling on Out-of-Order Processors*,” in Proceedings of the 30th Annual Symposium on Microarchitecture, 1997.

[2] Youfeng Wu, Yong-fong Lee, Hong Wang, “*An Efficient Software-Hardware Collaborative Profiling Technique for Wide-Issue Processors*,” Workshop on Binary Translation, Newport Beach, California, Oct. 1999.

Figure 4. SIDEBAR: An Efficient Software-Hardware Collaborative Profiling Method — Youfeng Wu (Intel)

MMX instructions aid multimedia code by performing multiple parallel operations on four 8-bit or two 16-bit operands. Intel added MMX to *x86* in 1996 in the *Pentium* family. Compaq's *FX!32* uses binary translation to execute *x86 Win32* applications on *Alpha/Windows NT* platforms[1]. To support MMX in *FX!32*, an *Alpha* code template was developed for each MMX operation. Actually two sets of such templates were developed, one for the *21164* and one for the *21264 Alpha* processor. Since the *21264* has its own multimedia extensions (MVI) for value saturation and packing, some templates were trivial. Other MMX operations, like 64-bit logical operations, are part of the *Alpha* ISA, and hence also have trivial templates on the *21164*.

A big decision was the physical location to which *x86* MMX registers would be mapped on *Alpha*. Although all *Alphas* have 32, 64-bit integer and 32, 64-bit floating point (FP) registers, the *21264* has instructions to move values between integer and floating point registers, while the *21164* does not. The *21164* must read and write values to memory in order to transfer them. Thus we considered three choices:

- Store MMX values in **FP** registers. Since values must be moved to integer registers to execute, this is bad for the *21164*.
- Store MMX values in *integer* registers. This leaves fewer integer registers for all translated code — even that with no MMX instructions.
- Store MMX values in memory. LOAD/STORE traffic is higher and more data cache misses occur.

We decided to store MMX values in **FP** registers since delivery of *21264*-based machines was beginning, and future memory optimizations could reduce the penalty on the *21164*. Further, intermediate results could be kept in integer registers by postponing updates to MMX registers (i.e. **FP** registers) until necessary.

To get a fast first implementation, we wrote simple subroutines, a style already in use for high-complexity *x86* string and **FP** operations. Translated code, (1) loaded MMX values into argument registers, (2) called the appropriate MMX routine, and (3) unloaded the result from the destination register. Unfortunately, overhead proved high. Execution time of *Photoshop FACET*, our benchmark, was worse than non-MMX execution.

To remedy this, we inlined MMX instruction translations instead of calling subroutines. The resulting *FACET* performance on a 500MHz *21264* was superb, beating a 266MHz *Pentium II* (3.77 versus 6.24 seconds.) Performance on *21164* lagged non-MMX execution. Measurement revealed the lost cycles were due to STORE/LOAD penalties.

[1] A. Chernoff, et al., "FX!32: A Profile-Directed Binary Translator," *IEEE Micro*, March/April 1998, pp. 56-64.

Figure 5. SIDEBAR: Adding MMX support to FX!32, — Paul J. Drongowski (Compaq)

legacy machine. If the new machine has fewer registers, then some legacy register values must be kept in memory with costly LOADS and STORES used to access them.

- System state stored in **special purpose registers** must also be maintained in the new architecture. For example, if the legacy architecture has special segmentation registers, they would have to be mirrored by the target architecture.

Flag registers which maintain condition codes must be dealt with. One simple approach is to have the new architecture set them in the same way as the legacy machine. Alternatively, a translator may be able to use a combination of redundant flag elimination plus smart flag calculation [9].

- **Atomic instructions** from the legacy architecture (e.g., synchronization primitives) should be replicated in the new architecture, while minimizing the semantic gap.

- **Non-interruptability** causes problems similar to atomicity. Some architectures such as the IBM *S/390* have complex instructions that take many cycles to execute and are *non-interruptable*, in that they must execute completely or not at all.

The new architecture can “pre-run” translations of such instructions without side effects to guarantee the “real run” will work.

- This *non-interruptability* problem is one instance of the more general problem of dealing with **precise exceptions** for the legacy machine. For example, if a particular LOAD causes a legacy alignment exception, the legacy exception handler will expect all instructions prior to that LOAD have completed and none after. Precise exceptions are harder still if translated code is re-ordered. The *BOA* paper in this issue discusses one method for dealing with this problem.

- Precise **discovery** of the legacy code is also a problem — especially for *static translation*. Given an executable file, it is not always clear what is code and what is data. The general problem of code discovery is undecidable, but various tools have used value tracking and other methods to solve this problem successfully in most practical situations.

- A problem related to finding legacy code is **self-referential legacy code**, i.e. legacy code which

looks at itself, for example to perform a checksum. Because of this, a copy of the legacy program counter (PC) must be maintained by the new machine, and used whenever the legacy machine references the PC for anything other than instruction fetch. Examples of such references include: PC-relative LOADs, PC save/restore on a subroutine call, etc.. Because legacy code can look at itself, the original code cannot be destroyed.

- **Self-modifying code** presents problems similar to *finding legacy code* and *self-referential legacy code*. Handling self-modifying code with a purely static translator is not possible, and is not always easy even with a dynamic translator — the new architecture must provide some means to detect when legacy code is modified, so that translations corresponding to the modified code can be invalidated.

A problem particular to *whole system* translators is **memory mapped** I/O. References to I/O locations can have side effects such as injecting a packet into a network or turning on an alarm, and must be done in program order and without caching.

4 Integrating Binary Translation into a System

The ultimate goal of binary translation is to provide the illusion of *transparency*: code can run on platform A, *exactly* as if it were run on a different platform B. Most of the aspects of transparency that we have discussed thus far relate to instruction-set conversion. But there are several other aspects:

- A user may wish to run an application that was compiled for one OS on a different OS. For example, we may want to run a *Windows* productivity application on a *Unix* workstation.
- Applications may be moved between different flavors of a single OS, as when transferring a *Win32* application to a 64-bit *Windows* environment.
- Translated applications may be cached in persistent storage, yet the normal semantics of an executable file must be preserved. When a legacy executable is changed, its translated equivalent should be invalidated.

- A legacy platform may provide instruction semantics that have to be emulated on a newer platform. For example, direct I/O instructions cannot be executed directly on the hardware in most modern OS's. Solving this issue typically requires emulation of legacy hardware devices.

4.1 Complete Operating System Emulation

There are several ways to deal with this OS emulation. One way to bypass most of the issues is to emulate the entire legacy OS. This approach results in a structure similar to a *virtual machine*.

The concept of a *virtual machine* dates back to the early 1960's [3], and has recently come back into vogue [7]. A classical virtual machine lets code run freely, as long as no privileged instruction is executed. For a privileged instruction, a special code sequence emulates the intended operation, possibly using primitives of an underlying OS. When integrated with binary translation, most code in a legacy OS would be translated normally, with special code sequences executed for privileged instructions. As modern virtual machines stabilize, we expect to see integrated solutions consisting of a binary translator interfaced to a virtual machine.

The approach of complete OS emulation is practical if, as in *DAISY* [2], the whole purpose of the new platform is to emulate a legacy platform (e.g., using a very fast CPU). This is not always possible since the cost of emulating the entire OS may be prohibitive for some systems.

4.2 Application-Level Translation

A more common solution than binary translation of the complete platform involves a port of the OS to the new architecture, with binary translation employed only at the application level. Thus, the new OS would run both native and translated legacy applications.

When the OS identifies a legacy executable, it launches the translator. Afterwards, most of the work can be done at the application level. Note that when static translation is used, the OS can still provide transparency by diverting execution from the legacy executable into the pre-translated native executable.

Even when an application is migrated between similar OS's, say from one *Unix* to another, several issues should be resolved at the application/OS interface:

- *Calling conventions*: one machine may pass parameters in registers and another may pass them on the stack.

- *Mapping of memory areas*: the OS may have trouble locating the application's stack.
- *Alignment of variables and structures in memory*.

The problems multiply when a translated application needs to communicate with a native application directly (e.g., through **D**ynamically **L**inked **L**ibraries), through shared memory or through another inter-program communication mechanism.

When the application needs to be migrated to a different platform, a "stub" or "jacket" layer may be employed to transform OS calls from the semantics of the legacy OS to the new one.

Proper exception handling is a difficult issue for application-level translators. Architectural compatibility and OS dependencies need to be carefully considered by the translator (e.g., the stack should be unwound and a correct machine state provided).

5 Conclusions and Further Research

We have outlined some of the major advances in the field of binary translation, described a number of common problems encountered in binary translation, as well as discussed how many of these problems can be resolved. However, we believe that improved techniques are still needed in a variety of areas:

- Improving translation from *virtual instruction sets* (e.g., the *Java Virtual Machine*).
- Providing *full-system translation* in which not only applications, but the entire operating system, is translated.
- Utilizing more sophisticated *profiling* techniques. An open question is whether changes in the program's behavior should be tracked continuously, or sampled just once — risking lower performance if behavior changes,
- *Bridging the semantic gap between architectures*, as described in Section 3. Further gains are needed to efficiently support, among other things, *non-interruptibility*, *atomicity* and support for multiple *legacy architectures* in one *target*,
- *Annotating binaries* to provide additional information to the translator beyond that given by the object code semantics. *Java* class files have moved in this direction, mostly to facilitate safety and security analysis, and

- *Developing new compilation techniques which benefit directly from dynamic code generation*, as was touched on in Section 2. Examples include aggressive speculation, procedure inlining and specialization.

References

- [1] Paul Drongowski, David Hunter, Moreza Fayyazi, David Kaeli and Jason Casmira, “*Studying the Performance of the FX!32 Binary Translation System*,” IEEE TCCA Newsletter, December 1999, pp. 56-68.
- [2] Kemal Ebcioglu and Erik R. Altman, “*DAISY: Dynamic Compilation for 100% Architectural Compatibility*”, In Proceedings of the ISCA-24 pp. 26-37, Denver, CO, June 1997. ACM.
- [3] R. Goldberg, “*Survey of Virtual Machine Research*,” IEEE Computer 7(6), pp. 34-45, 1974.
- [4] Raymond J. Hookway and Mark A. Herdeg, “*Digital FX!32: Combining Emulation and Binary Translation*,” Digital Technical Journal, Vol. 9, No.1, 1997, pp. 3-12.
- [5] David Kaeli (editor), “*TCCA Newsletter*”, December 1999, Proceedings of 1999 Workshop on Binary Translation, Newport Beach, CA.
- [6] John Kalamatianos, Alireza Khalafi, David Kaeli and Waleed Meleis, “*Analysis of Temporal-based Program Behavior for Improved Instruction Cache Performance*,” IEEE Transactions on Computers , February 1999, pp. 168-175.
- [7] M. Rosenblum, S. Herrod, E. Witchel and A. Gupta, “*Complete Computer Simulation: The SimOS Approach*,” IEEE Parallel and Distributed Technology, Fall 1995.
- [8] Richard Sites, Anton Chernoff, M. Kirk, M. Marks, and S. Robinson, “*Binary Translation*,” Digital Technical Journal, Vol 4, No 4, Maynard, MA.: DEC, 1992.
- [9] Murari Srinivasan, “*Method and Apparatus for Emulating Status Flag*”, U.S. Patent 5774694, June 30, 1998.

Erik Altman is in the High Performance VLSI Architecture Group at IBM T.J. Watson Research, and has worked on the DAISY and BOA binary translation

projects. Altman has an MS and PhD from McGill University. Contact him at erik@watson.ibm.com.

David Kaeli received his PhD in Electrical Engineering from Rutgers University. He is presently an Associate Professor on the ECE faculty at Northeastern University. Prior to Northeastern, Prof. Kaeli spent 12 years at IBM. He is the Director of the Northeastern University Computer Architecture Research Laboratory (NUCAR). He can be contacted at kaeli@ece.neu.edu.

Yaron Sheffer received his BA in Computer Science from the Technion, and his MSc from the Hebrew University in Jerusalem. Yaron spent eight years at Intel, leading a binary translation effort. Recently Yaron joined Radguard, a network security company, where he researches technologies. He can be reached at yaron.sheffer@radguard.com.