DELI: A New Run-Time Control Point

Giuseppe Desoli, Nikolay Mateev, Evelyn Duesterwald,
Paolo Faraboschi and Joseph A. Fisher

Hewlett-Packard Laboratories, 1 Main St., Cambridge, MA 02142, USA
giuseppe.desoli@st.com, nikolay.mateev@hp.com, duester@us.ibm.com,
paolo.faraboschi@hp.com, josh.fisher@hp.com

Abstract

The Dynamic Execution Layer Interface (DELI) offers the following unique capability: it provides fine-grain control over the execution of programs, by allowing its clients to observe and optionally manipulate every single instruction—at run time—just before it runs. DELI accomplishes this by opening up an interface to the layer between the execution of software and hardware. To avoid the slowdown, DELI caches a private copy of the executed code and always runs out of its own private cache.

In addition to giving powerful control to clients, DELI opens up caching and linking to ordinary emulators and just-in-time compilers, which then get the reuse benefits of the same mechanism. For example, emulators themselves can also use other clients, to mix emulation with already existing services, native code, and other emulators.

This paper describes the basic aspects of DELI, including the underlying caching and linking mechanism, the Hardware Abstraction Mechanism (HAM), the Binary-Level Translation (BLT) infrastructure, and the Application Programming Interface (API) exposed to the clients. We also cover some of the services that clients could offer through the DELI, such as ISA emulation, software patching, and sandboxing. Finally, we consider a case study of emulation in detail: the emulation of a PocketPC system on the Lx/ST210 embedded VLIW processor. In this case, DELI enables us to achieve near-native performance, and to mix-and-match native and emulated code.

1. Introduction

This paper is about a new capability in computing systems, one that is subtle, and one that we believe can have a large effect on computing. We are all familiar with the method of loading and executing programs, where most of the traditional program transformations terminate before the program binary runs. What has changed in the last decade is the steady growth of techniques that continue manipulating code while the program is running, from dynamic loaders to *Just-In-Time* compilers. Some of these techniques share a fundamental property: they observe—and potentially transform—instructions of the target program immediately before they run.

System utilities that operate on *programs* as their target datasets have many different motivations. Sometimes, the semantics of the target program are not meant to directly address the hardware on which it runs, and we use compilers, interpreters and emulators to translate the target program and perhaps initiate its execution. Linkers and loaders process the target program in order for it to run correctly. Some tools check a target program for viruses or other properties, while others, such as profilers, measure performance-related properties of the target.

Compile time vs. run time. Some programs process a target binary before execution, and then get out of the way, so that the target program can run. We often say that these transformations happen at *compile time* (or load time). Other approaches operate on a binary while it is running, and we sometimes say they operate at *run time*. Compilers, unsurprisingly, are examples of the first type, as are most virus checkers. Classical emulators, which fetch each instruction of the target, translate it, and then initiate execution, are examples of the second. Superscalar control units are another example of run-time transformation, as they schedule instructions just before they run.

Persistent vs. transient changes[1]**.** A similar consideration has to do with whether changes to the same part of the program have a long lifetime, or whether their effect only lasts briefly. For example, compile time techniques operate in advance, we think of them as making a single change to the program, and we call them *persistent*. Most run time techniques, however, change the same parts of the program repeatedly, and we thus refer to these as being *transient*. Classical emulators and superscalar hardware both touch every single instruction repeatedly, immediately before its execution, and the result of their transformation is transient in this sense.

Persistent changes at run time. It might be fair to refer to

[1] Many would refer to this distinction as *static vs. dynamic*. Unfortunately, *dynamic* is also a synonym for run-time. To avoid confusion, we adopt *persistent* and *transient.*

Proceedings of the 35 th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-35)
1072-4451/02 $17.00 © 2002 IEEE

the actions of a dynamic loader as persistent, since the effect of the changes is repeatedly used. Much more important is a technique that emulators widely apply: *caching of translated code*. To avoid repeating the translation of the same piece of code, emulators relocate chunks of translated code into a private cache. As long as execution stays in the cache, this avoids both the cost of the translation and the cost of invoking the emulator. When done well, in our experience it yields savings up to 100x. We normally would think of this technique as being persistent, but the work is being done at run time[2]. In some sense, this type of mechanism combines the advantages of compile time and run time. We can amortize work over a long time, yet we take advantage of the near-perfect knowledge available at run time. With today's late-binding coding style, this advantage is becoming ever more important.

Table 1 - Although we usually identify run time execution with dynamic (transient) changes, it is useful to separate the concepts of what is done at run time vs. compile time and what is used once or used repeatedly.

	Compile Time	**Run Time**
Persistent Changes made once, used many times	Compilers	Dynamic loaders (and DELI)
Transient Changes made many times, used once	*[Doesn't make sense]*	Superscalar hardware

1.1. Effect of a new control point

Efficient emulation already implies a lot of power: even though code was produced for a stipulated instruction-set architecture, code still runs correctly despite ISA changes. Now imagine a much more flexible capability as follows.

- A system tool that gives clients ultimate fine-grain control over programs running on the system by allowing the client to observe and optionally manipulate every single instruction in the target—at run time.
- This system is guaranteed to be the last piece of software to touch an instruction (code runs out of a cache, and the system observes and potentially manipulates the code before it is placed there).
- The system accomplishes this by opening up an interface to the workings of a native-to-native binary emulator, which uses caching and linking to stay close to native performance.

With a capability like this, it is possible to have a view of object-code compatibility and many other issues which, together, can dramatically change some of our assumptions about computing. Just as superscalar hardware makes transient rearrangements of code at run time to match it to ILP hardware, software can make persistent code changes at run time to match the object code to the hardware that is actually present when the code runs. This facility could free up the microarchitect in many ways, raise the level of processor compatibility above the hardware level, facilitate software migration, and allow us to design hardware that does not have to pay a price for compatibility every single execution cycle. While many emulation systems offer the same freedom, DELI does so in a way that is flexible, easy to use with multiple versions of the hardware, and not requiring large re-implementation efforts. It also enables us to examine and manipulate code in many other ways, in the perfect light of run time, but with the cost of the necessary analysis and transformations amortized over the full period of the persistent use of the result.

[2] This has been referred to elsewhere as *Walk Time Techniques* [16]: you're moving, but you're not in a frenzy.

1.2. Related work

The DELI provides a uniform infrastructure for building client applications that manipulate or observe running programs. An important example of this type of application is *emulation*. Advanced emulation systems use varying degrees of code caching and optimization ranging from caching unoptimized individual translated code blocks [7] to sophisticated dynamic binary translation systems [9][11][24][26][28]. Examples include the Daisy binary translation system [11], Transmeta's *Code Morphing* software for the Crusoe processor [9] and Transitive's *Dynamite Software* [24]. These systems provide a complete software layer for dynamic binary translation between different ISAs, where the mechanisms for code caching, linking and optimization are an integral part of the overall system. In contrast, the DELI explicitly isolates the code cache functionality in a separate software layer.

The DELI is not an emulation system itself: it encapsulates the common code caching and linking functionality, which can then be leveraged across a number of different emulation systems. The DELI shares the capabilities for dynamic code optimization with binary translation systems. However, while in binary translation systems, optimization is tightly integrated, the DELI offers it as a service. Thus, it frees the emulation system developers form designing and implementing target-specific optimizations.

We view dynamic optimizers [1][6][8][25] as native-to-native binary translation systems, where performance improvement is the sole desired effect of translation. Mechanisms for dynamic hot spot detection [22] and dynamic optimization [17] have also been implemented in hardware. As in dynamic optimizers, the DELI may employ code optimization transparently to accelerate native binaries. However, the DELI goes beyond that by specifically opening up the dynamic optimization functionality as a service to client applications. Under specific instructions from the client, the DELI enables dynamic code optimizations that are beyond the reach of a purely transparent dynamic optimizer, whose code knowledge is limited to the binary text.

The Trace Cache [17][25] is a hardware approach to optimize the memory bandwidth of running programs. Unlike the DELI, hardware mechanisms are not easily extensible and are completely hidden from application and system software.

Java Virtual Machines (JVM) with Just-In-Time (JIT) Compilation [21] or the *Common Language Runtime* in the Microsoft's .NET environment [23] are advanced runtime systems for executing portable intermediate code (i.e., Java Bytecode or Microsoft Intermediate Language). These higher-level emulation systems use dynamic caching of translations (a.k.a. *JIT compilation*) for performance. A JVM (or Common Language Runtime) provides a similar level of control over the execution of the intermediate code as the DELI achieves for the execution of binary code. In contrast, the DELI is language independent, does not require a special code format and can therefore even handle legacy code.

Runtime interfaces have been developed for specific runtime code modification tasks. The *DynInst* API [4] and the Vulcan system [27] can be used to insert instrumentation code into a running program. *DynamoRIO* [10] (based on Dynamo [1], which is also a DELI ancestor) is a dynamic optimization system that exports an interface to implement arbitrary code transformations while the program executes. Like the DELI, these systems export an API to higher-level application clients. However, unlike DELI, these systems are designed for and limited to a specific kind of native-to-native dynamic code modification

Finally, *Debugger APIs*, though their goals are very different, are similar to the DELI in that they open up a well-defined interface to the operation of programs at this same low level. As with the DELI, programs (debuggers) use this interface to supply services that probe the running program. Unlike the DELI, Debugger APIs are written with one narrow purpose in mind, and don't offer access to the low-level operation of programs for varied use. For example, a debugger is normally a separate process, not concerned with the performance of the debugged program.

2. Overview of the DELI System

The DELI is a software layer that operates between application software and the hardware platform as depicted in Figure 1. Depending on the desired functionality, the DELI layer can be inserted underneath or above the operating system. For example, if system code should execute under DELI control, the DELI would be inserted underneath the operating system.

Despite its novelty, the DELI is an industrial-quality tool. It is robust enough to support complex operating systems (like WindowsCE), and can deal with all the nuisances of real-world systems, such as system calls and self-modifying code.

To understand the role of the DELI with respect to its clients, it is helpful to consider an analogy with operating systems. The DELI is to its client application what virtual memory is to an ordinary application. Strictly speaking, building an application does not require virtual memory or other operating systems support, and in some deeply embedded domains applications often embody significant portions of the operating system functionality. However, the presence of an underlying operating system greatly simplifies application development. Similarly, the presence of the DELI can greatly facilitates the construction of dynamic code transformation functionalities in client applications.

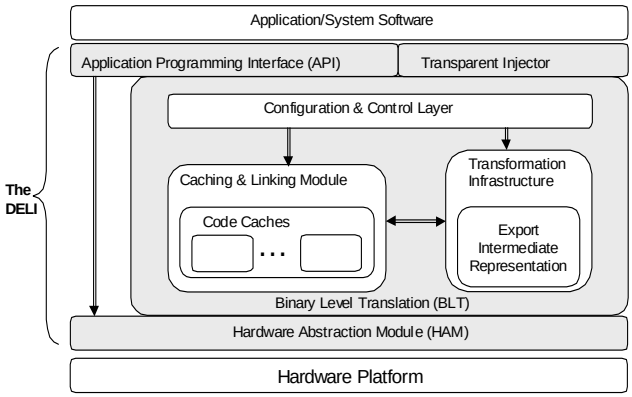

Figure 1 - An overview of the DELI system

As shown in Figure 1, the DELI layer includes three main components: the *Binary Level Translation* (BLT) layer, the *Hardware Abstraction Module* (HAM), and the *Application Programming Interface* (DELI API). The BLT layer provides the core code caching and linking functionality, and it includes several code caches and basic infrastructure elements for binary code transformation, such as optimization and instrumentation. HAM provides virtualization of the underlying hardware. The DELI API makes the functionality accessible to client applications. We will discuss the role and functionality of each component in detail in the following sections.

2.1. The DELI API

Through its API, the DELI provides basic code caching and linking service, as well as the necessary infrastructure to support dynamic code transformation to the running client applications. To illustrate the DELI API, consider the following scenario: a client emulation system that is taking advantage of the DELI to cache and link translations of the emulated code. To avoid repeated emulation of the same code sequence, the emulation system produces a code fragment that contains a translation of the code sequence. The emulation system uses the DELI to emit the fragment for caching and linking via the API function:

```
deli_emit_fragment (tag, start, end,
                    flags, user_data)
```

Table 2 - The DELI core API

DELI API Function	Description
`void deli_init();`	Initialize the DELI
`void deli_emit_fragment(tag, start, end, flags, user_data);`	Emit a code fragment in the DELI code cache. Arguments include a client-level tag identifier, the location of the translation (start and end address), flags and user_data. The flags specify attributes of the emitted fragment, such as the optimization level to apply, or whether the fragment is instrumented for profiling.
`void deli_exec_fragment(tag, context);`	Start executing client code from the location identified by tag with the given machine context. Arguments include the fragment's tag identifier and a machine context that the client must load before executing the fragment
`deli_handle deli_lookup_fragment(tag);`	Return a DELI internal handle for the fragment identified with tag, if it currently resides in the cache
`void deli_invalidate_fragment( handle);`	Invalidate the fragments identified with the DELI internal handle.
`void deli_install_callback( deli_event, callback);`	Register a client-level callback function to be invoked upon a specific DELI event, such as fragment emission, cache exit, profile counter overflow, or out-of-space code cache condition. We can use the callback mechanism to implement a variety of tasks, such as enforcing client-level code assertion, controlling code cache flushes, or triggering optimization and reformation of fragment code once it becomes hot (when the profile count exceeds a threshold).
`void deli_enum_fragment( callback);`	Apply a client-level callback function to each fragment currently residing in the cache. Fragment enumeration is useful, for example, to enforce specific checks or instrumentation code on all fragments.
`cache_object_id deli_setup_cache(cache_obj_desc, flags, user_data);`	Create a code cache object within DELI with given attributes (size, priority, policies) and properly links it in the DELI code cache stack.
`void deli_code_cache_flush( cache_obj_id, flags, timeout);`	Initiate a DELI code cache flush on the given code cache, with a given timeout.
`int deli_gc(cache_object_id, flags, timeout);`	Initiate on-demand garbage collection on a given code cache, with a given timeout
`void deli_start();`	Start DELI "transparent" mode, where DELI takes control of the running program
`void deli_stop();`	Stop DELI "transparent" mode

The next time the emulation system is about to emulate the code just translated, it can instruct the DELI to execute the fragment with:

```
deli_exec_fragment(tag, context)
```

Internally, the DELI directly interconnects all emitted fragment code whenever possible. Thus, invoking `deli_exec_fragment()` may actually result in the execution of a sequence of fragments until we encounter a fragment exit that is not connected, in which case an exit tag will be returned to the client. Table 2 shows a description of the most important functions of the DELI API.

Besides implementing the API, the DELI is also capable of acting in a transparent mode with respect to the client application. In this mode, the DELI transparently takes control over the running client application, such that it operates like a native-to-native caching emulator similar to the HP Labs Dynamo dynamic optimizer [1]. The DELI intercepts the running applications and copies every piece of code that is about to execute into its code cache, so that execution only takes place out of the code cache. When placing code into the code cache, the DELI has the opportunity to optimize it, as in the Dynamo system. However, unlike Dynamo, the DELI can be instructed through the API to perform other code transformation or observation tasks beyond optimization (such as instrumentation or insertion of safety checks). An application client can explicitly turn on and off the transparent mode by invoking `deli_start()` and `deli_stop()`. In addition, the DELI is equipped with a transparent injector module; if configured correspondingly, the DELI automatically injects itself and takes control over the application. Automatic injection is useful for implementing code-caching services for applications that are not DELI-aware.

2.2. Binary Level Translation (BLT)

The BLT layer is the core component of the DELI, responsible for implementing the DELI API. As shown in Figure 1, the BLT layer contains and manages a set of code caches. It also provides the basic infrastructure for code transformations. DELI obviously introduces overhead, and the key to amortizing it is to ensure that the application code runs efficiently inside the code caches. The DELI's primary means for delivering code cache perform-

ance are *fragment linking* and *dynamic optimization*.

Linking Fragments

Linking avoids unnecessary code cache exits by directly interconnecting the fragments of code, and it requires matching the fragments' exit and entry tags. Recall that fragment tags are created by the client when emitting code through the API: DELI assigns each fragment a unique entry tag and marks each fragment exit with a corresponding exit tag. Upon fragment emission, the DELI inspects each fragment's exit tag to directly interconnect it to a corresponding fragment entry. If no such entry can be found, DELI temporarily redirects the fragment exit to a special *trampoline* code. The trampoline causes execution to exit the code cache and to eventually return to the client with the missing exit tag.

The above strategy takes care of creating a fragment's outgoing links. To establish incoming links, the DELI uses a deferred strategy. Initially, a fragment is entered into the code cache without creating direct incoming connections. At each code cache exit a test is made to determine whether a fragment for the exiting target tag has been materialized in the meanwhile. If so, a direct link is established, and execution continues in the code cache instead of returning to the client.

The Transformation Infrastructure

An important feature of DELI's BLT layer is the capability to dynamically optimize the emitted code. The BLT layer provides a complete dynamic optimization infrastructure to activate runtime code optimization, either explicitly through the API, or autonomously by setting the appropriate optimization policy.

The core of DELI's transformation infrastructure is the DELI *Intermediate Representation* (DELIR). DELIR is a low-level intermediate representation that maintains a close mapping to the underlying machine representation while providing sufficient abstraction for permitting code motion and code relocation. DELIR serves two purposes: to internally enable the transformation of fragment code, and to facilitate the construction of code fragments in the client. For the latter, DELIR is exported to the client through an extension of the DELI API. In addition to emitting a fragment in machine code representation, the client can construct a DELIR fragment and pass it to DELI for code emission. Table 3 shows a sample of the DELIR functionalities.

By instructing the DELI through the API, we can apply optimizations to a fragment either the first time the fragment is emitted, or when it has become "hot" in the code cache. DELI considers a fragment "hot" when it exceeds an execution threshold that clients can set through the DELI API (default is 500 executions). When a fragment exceeds the threshold in the cache, the DELI decodes the fragment code to produce a DELIR fragment. DELI then passes the DELIR fragment to the internal lightweight runtime optimizer and scheduler. Depending on the chosen optimization level, the runtime optimizer performs up to two passes over the code: a forward and a backward pass. During each pass, it collects data flow information *on the fly* as the pass continues either forward or backward. The particular kind of optimizations and the particular scheduling algorithm used depend upon the underlying machine characteristics. Our current prototype, detailed in Section 4.1, targets an embedded VLIW architecture, so the corresponding set of optimizations that we currently employ in DELI is targeted towards increasing ILP: copy propagation, constant propagation, dead code elimination and strength reduction. The complexity of the optimizations is tunable by a peephole window. The performance benefits of these optimizations are primarily in improved scheduling quality.

Table 3 - The DELIR API

DELIR API Function	Description
`fragment* delir_create_frag( int id, unsigned flags);`	Create a DELIR fragment, where id serves as a fragment identifier and the flags are used to specify various fragment properties. The function returns a handle to the created fragment.
`void delir_append_inst( fragment *frag_handle, inst *inst_handle);`	Append an instruction (specified by the handle inst_hande) to the DELIR fragment specified by the handle frag_handle.
`inst* delir_make_opcode( int opcode, unsigned *arglist, unsigned flags);`	Create a specific DELIR instruction with the specified opcode, the arguments that are passed in the arglist and the specified flags. The function returns a handle to the created instruction.

A large number of static scheduling algorithms targets VLIW scheduling (such as Trace Scheduling [15]). Unlike static schedulers, dynamic scheduling algorithms give highest priority to speed and a simple design in order to be competitively applied at runtime [11][26][28]. DELI employs two lightweight scheduling algorithms: an *instruction scheduler* and an *operation scheduler*. The schedulers do not require building a data dependence graph, and can be tuned by changing the size of the look-ahead window.

The **instruction scheduler** is a cycle-by-cycle scheduler, and computes the required dependencies on the fly within a fixed size look-ahead window, similarly to the way in which superscalar hardware operates.

The **operation scheduler** traverses the code one operation at a time in an attempt to schedule each instruction in the earliest cycle. This is similar to the scheduling algorithm in the Daisy system [11].

Section 4 provides detailed information on the performance of DELI's optimizations and scheduling.

2.3. Hardware Abstraction Module (HAM)

DELI clients may require low-level control of hardware events (such as exceptions) to do their job efficiently. In addition, we can make DELI itself more portable and resilient to changes in the underlying hardware platform if we adopt some form of abstraction from a specific platform. Modern OSs use a similar functionality, where a low level layer deals with all the platform dependent ways of coping with hardware related features, such as enabling/ disabling/dispatching interrupts, flushing the caches, and managing the TLBs. In the context of DELI (especially when it is below the OS), we need a similar piece of functionality, which needs to be separated from the equivalent OS layer. If we can extend or change the OS, then both the DELI and the OS could share the same layer.

From the point of view of DELI clients, it is useful to open up parts of this interface. For example, an emulator needs to efficiently emulate the virtual memory (VM) system of the original CPU, by mapping it onto the VM of the target system. To do this, the client needs access to the details of the hardware TLBs, needs to maintain a data structure (similar to a page table) for the emulated system, and needs to match it with the page table information of the native system. Only then can it service the various TLB-related events. This method has many disadvantages. The hardware TLB may change from one implementation to another, even within the same CPU family. Accessing the TLB though the OS interface that maps pages may not be expressive enough, for example for security reasons. Even when we can do this, it requires control over the OS to make the two VM systems properly overlap.

An alternative is to provide a 'virtualized view' of the hardware for both the OS and the DELI clients so that they can share the same representation. The HAM layer thus is built around a configuration infrastructure that has both static and dynamic components. The static configuration includes:

- A description of the 'fixed' memory mappings between address ranges. This could be used by a virtual machine implementation to describe the relationship between the original address in the emulated system and the addresses used to host that memory in the native world (see Table 4 for an example).
- A description of globally defined hooks for various events, such as exceptions and interrupts, that clients might require, along with the requested action that HAM has to take upon return from those hooks. Actions include skipping the excepting instruction, nullifying the memory operation, and so on.

The last three fields in the static configuration entry for a region (see Table 4) can be used to describe a region that needs a 'special' treatment, meaning that HAM can be set up to invoke a user defined hook whenever a memory access hits that region. This simplifies the task of virtualizing a piece of memory-mapped hardware, or can also provide an equivalent functionality with a different piece of hardware (e.g. a similar, but not completely compatible, peripheral). In this way, an emulator can describe a memory-mapped peripheral or device as a collection of functions to be invoked through those hooks. Internally, HAM maps this into the relevant protection attributes for translation entries that are eventually loaded onto the hardware TLB. Then, certain types of accesses can raise an exception that HAM intercepts to invoke the proper hook. In a binary translator, this relieves the translated code from the burden of guarding each memory access to determine whether we have to treat it specially.

Table 4 - Example of HAM configuration entry. For example, the entry *{1, 0xA0000000, 0x4000000, PREF_MMU_PAGE_SIZE, ACCESS_RIGHTS, NULL, NULL, 0, 0x20000000}* defines a region of (normal) memory for which HAM automatically and transparently handles all TLB misses starting at virtual address space 0x4000000 and mapped onto address 0xA0000000 with a size of 64MB in virtual address space 1 (the emulated virtual space).

Entry	Description
`space_id`	Identify this entry to be in a given virtual address space. We can use this to make multiple virtual address spaces coexist. HAM maps this into what the actual hardware supports (e.g., ASIDs or address space descriptors). It can be used to make an OS share the underlying VM with an emulator client, since they would be using different virtual space identifiers, when the emulator impersonates the emulated program.
`start_addr`	Start logical address of this region
`offset`	Start physical (or logical in certain cases) address of this region on the native system encoded as an offset from the previous.
`size`	Size of this region
`page_size`	Log_2 of the page size to be used to map the region's pages, used by HAM to guarantee the correct behavior, as if the pages were of this size. For efficiency HAM can eventually map pages onto different sizes.
`attributes`	Access rights and properties for this region
`load_hook`	Client-defined hook to be invoked on reads from this region (for 'special' regions)
`write_hook`	Pointer to a client-defined hook to be invoked on writes from this region (for 'special' regions)
`inv_state`	Determine what machine state needs to be installed before invoking the hooks. A client can use this to guarantee that a hook doesn't run in a higher privilege than the code issuing the load/store (for 'special' regions)

The rest of the HAM configuration is dynamic and it mainly manages the HAM page translation tables (hash tables augmented with information coming from the static mappings for regions). HAM usually deals with low-level

events (like TLB misses) transparently, but it sometimes has to invoke client-installed hooks.

The HAM interface is quite extensive and a complete description is beyond the scope of this paper. The DELI's BLT uses part of HAM to manage hardware related needs, and, at the same time, HAM relies on the DELI API for some of his advanced functionality. For example, HAM uses the API for the emission of temporary fragments to materialize and execute transient trampolines, when clients require a specific exception behavior.

3. Using DELI as a client

The DELI provides two types of services to client programs. First, DELI's caching and linking allows very easy development of efficient emulators of completely different ISAs and OSs. Second, DELI opens a new and very powerful control point to client programs. Clients can observe and potentially manipulate every single instruction of the target program immediately before that instruction runs. We find it useful to break down the many applications that benefit from DELI into three basic classes.

Code manipulation: clients that want to adjust and modify natively compiled programs at run-time. Examples include optimized software patching, code decompression and code decryption.

Observation and control: clients that want to observe the behavior of natively compiled programs at run-time and possibly enforce policies. Examples include sandboxing, dynamic virus detection, reporting program behavior, profiling, and statistics gathering.

Emulation: clients that want to run an application that was compiled for a different virtual machine, a different ISA, or a different hardware configuration. Examples include ISA interpretation or emulation of virtual machine environments, such as Java or .NET.

In the case of the first two classes, the manipulation and observation are done by the client program as the code is placed in DELI's cache. Since code is always run out of the cache, and nowhere else, these clients are guaranteed to be the last to see the code. DELI is a new concept, so not many applications have been written for it. In the rest of this section, we only cover a few examples of each category, to demonstrate the DELI's capabilities that allow for easy development of client programs.

3.1. Code Manipulation

Since DELI is the last piece of software that observes an instruction right before it is executed, it can replace the observed instruction with another sequence. All manipulation clients rely on DELI being a control point in the execution; DELI caches and optimizes the adjusted code, so it manages to keep the overhead to a minimum. Figure 2 shows the common scheme for code transformations: clients provide a *manipulation routine* through the `deli_install_callback()` API call that DELI dynamically applies during fragment formation. The routine transforms the code that is then emitted to the DELI code cache for efficient execution or re-optimization.

In the following, we present two manipulation examples: dynamically patching code and dynamic code decompression (or decryption).

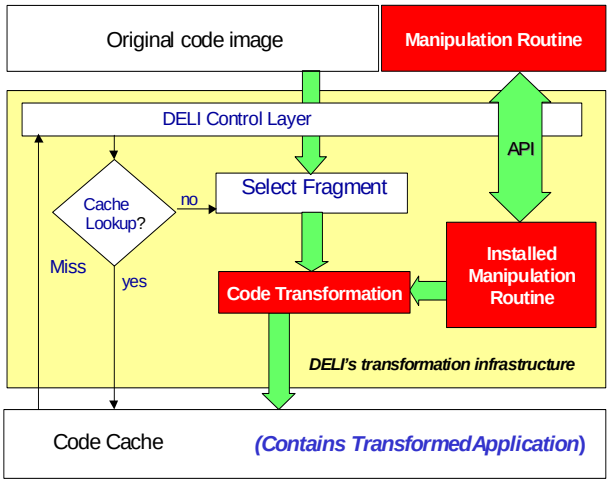

Figure 2 - Using DELI to dynamically manipulate code (e.g., for patching or decompression). The diagram expands the "*DELI transformation infrastructure*" component of Figure 1 and shows how a client can install a *manipulation routine* through the DELI API for code transformations.

Dynamically patching code

Suppose you have faulty or missing hardware in your system—for example, your processor's floating-point unit is faulty, or your *x86* processor has no MMX functionality. Traditionally, solving such problems implied either replacing the faulty (or missing) hardware, or rewriting software so that it does not require the faulty or missing functionality. Neither of these solutions is particularly attractive. Replacing hardware is expensive and inconvenient. Traditional software solutions require rewriting, recompiling, or applying static code patches, involve permanent changes to the software, and can't easily deal with third party components.

DELI gives us the opportunity to patch software dynamically and temporarily, while the original software remains untouched. DELI observes each instruction right before it is executed, so it can detect which instructions are affected by the missing hardware functionality, and dynamically replace them with new code that does not require it. The *manipulation routine* of Figure 2 for this transformation involves maintaining a patch table, with a patch descriptor for each different type of patch request. Patch descriptors include an identifier of the missing hardware and a code sequence that works around the issue.

Code decompression or decryption

Firmware memory amounts to a significant fraction of the cost of an embedded system, so *code compression* [30] is a commonly used technique to improve the efficiency of program storage. In most schemes, a post-link software utility compresses the code at compile time. In systems with hardware decompression, a dedicated unit expands the code during instruction cache refill, on a cache line-by-line basis. Traditional software-based decompression schemes handle large chunks of code at a granularity that matches the operating system structures (usually at the level of file systems or pages). Hardware schemes are inflexible and expensive, and traditional software schemes carry a large performance overhead when we only need a small fraction of the decompressed code.

DELI can help to avoid both problems, as we can use it to decompress code fragments as they get executed, and still achieve good performance by caching the most frequently used code. Referring to Figure 2, in this case the *manipulation routine* implements the decompression algorithm: the transformed code is really the original program code before compression. In this way, the DELI only decompresses and caches smaller portions of the application, thereby reducing overhead and amount of dynamic decompression, and requiring less system memory.

Note that this solution also extends to other forms of coarse-grain program transformation. For example, in security-aware domains, it is common to sign and encrypt applications so that accesses to the application can be trusted and controlled. Here, we can use exactly the same scheme that we described for decompression, where we substitute a *decryption routine* for the decompression routine, achieving exactly the same benefits: using the DELI is cheaper and more flexible than a hardware solution, and more efficient than a traditional software approach.

3.2. Program Observation

The fine-grain control that DELI provides is also useful when we need to report program behavior, or enforce certain policies. Using a scheme similar to what we described in section 3.1, a client can install an *observation routine* through the API, cause DELI to examine the code at run time, and only emit it to the cache if it respects the policy to be enforced. This does not require transformations (beyond instrumentation), and the overhead is minimal.

Sandboxing

In a multiprogrammed environment, multiple applications share a given computer system. Execution "sandboxes" [5] are environments that support differentiated services for the different applications and impose restrictions on resource usage. Resource restrictions can be qualitative (e.g., permitting the application to access only certain memory addresses, or certain portions of the file system), and quantitative (e.g., limiting the CPU usage of an application to 25%). Existing sandboxing approaches include hardware solutions, such as virtual memory for memory protection, OS kernel modifications, and interception of system calls.

With DELI, we can install an observation routine that examines each code fragment before it is emitted into the code cache. That routine verifies the legality of the instructions (e.g., checks that direct memory references access only the memory range assigned to the application), and instruments the code fragment to self-check other instructions that cannot be statically verified. Once the code is modified and cached, it self-checks its legality during execution without the need to interrupt the application. Moreover, with profiling instrumentation [2] of the code fragments before they are emitted into the code cache, clients can also monitor and enforce quantitative resource usage policies.

In hardware platforms with security support, DELI itself has to run in *trusted mode* (at a privilege level above supervisor) to ensure the overall security of the system.

3.3. Emulation

With DELI in place, we can imagine building the software infrastructure (and possibly efficient hardware) to implement a "universal core" for embedded applications. This system could pair emulation of an existing ISA with natively compiled high performance kernels. For example, we could build a VLIW engine and add an ARM software emulation layer on top of that. This would enable users to preserve their investment on existing ARM code and tools, and gradually migrate to native VLIW code only the parts that are more relevant for performance. The existence of DELI (and possibly a set of DELI-aware hardware features) ensures that the emulation overhead is kept to a minimum. For developers, this means a gradual migration path, performance scalability at a reduced engineering cost, and vendor independence. In other words, it diminishes the impact of legacy code.

By using the DELI infrastructure, we are developing a technology that includes emulation modules for legacy operating systems, an interface to mix native and emulated code, and a set of techniques for "system emulation" (i.e., to emulate differences in areas such as interrupt control and virtual memory).

Code streaming

An interesting application of the DELI native-to-native emulation capabilities (when emulated and target ISAs coincide) deals with the situation in which parts of the original code live in a remote location with respect to the target machine. The DELI opens up an alternative model for remote execution, that we call *code streaming*, where the remote application is loaded on-demand, one fragment

at a time, based on its execution paths.

The code-streaming client simply instructs the DELI to emit fragments in the code cache, and regains control when execution encounters remote program locations. If we again refer to Figure 2, the client *manipulation routine* is here responsible for accessing the remote application, and for keeping the correspondence between local and remote program locations. The DELI caches the application locally, so, unlike more traditional remote execution models, it can tolerate an intermittent connection with the server. What is special about code streaming is that applications do not need to be aware of its existence. DELI's *transparent mode* does not rely on the use of a specific API and it fits well with legacy applications. It is also OS-independent, does not require proprietary file systems, and does not rely on a specific language (unlike Java or .NET).

4. Case Study: Emulating PocketPC

This section presents an application of the DELI to the world of *mobile multimedia* devices. Mobile communication devices have grown to a level of maturity that allows deployment of rich web and media-enabled applications, thus demanding a significant increase in system complexity and performance requirements. However, embedded operating systems have consolidated to only a few competitors, which are only going to support a restricted number of different ISAs. In this context, we can see *emulation* as a potential solution to some of these barriers to innovation, opening the way to new architectural features. We can identify three main scenarios where efficient emulation of an embedded ISA would be beneficial.

Emulated platform with native media engines. Emulation enables developer to leverage legacy applications and GUIs (such as *PocketPC*), while combining them with high-performance media engines natively compiled to a more powerful processor. This model requires an *emulated-to-native* interface like the one DELI provides.

Native platform with emulated plug-ins. Even for entirely new platforms (e.g., embedded Linux devices), there is still a strong need to support legacy applications, in the form of *plug-ins* (e.g., a media viewer). Similar considerations apply Java or .NET environments for efficiency.

Incremental migration. For families of binary incompatible processors (e.g., many VLIW embedded offerings), the problem of legacy code compatibility across different members of the same family could be efficiently tackled by the use of a lightweight dynamic translator. The DELI provides the building blocks that can be used to build such hybrid systems, so that we can guarantee a smooth migration between successive generations of processors.

All these scenarios require mixing emulated and native code at different levels; DELI efficiently tackles all of them, by exposing the desired level of granularity that best meets application needs at the programmer's level. For example, Original Equipment Manufacturers (OEMs) could use a coarse level of granularity to provide native multimedia libraries integrated in the context of a standard component of the emulated system (e.g., *DirectX* drivers in PocketPC). Alternatively, application developers could leverage the underlying capability of the native core by providing their own native implementation of the media kernels, still within an existing OS and development environment. Finally, if we see the DELI as a way of providing an efficient virtual machine environment, this can open up new opportunities to extend system functionality with no changes to the OS.

4.1. The DELIverX prototype

DELIverX is the prototype system that we built to demonstrate the feasibility of this approach and at the same time to help us better understand the details of the DELI programming interface. The DELIverX emulation system is defined around a *Hitachi SH3* interpreted emulator, coupled with a just-in-time translator for an embedded VLIW core, which we use as a native target. DELIverX implements a platform emulator (i.e., ISA + system) that can be used as a target for a Microsoft PocketPC operating system compiled for the SH processor. The SH-3 [2] is a simple load-store RISC processor with a 5-stage pipeline that executes most basic instructions in one clock cycle. The SH3 implements a 16-bit ISA but fetches two instructions in a single access. The target processor is the Lx/ST210 4-wide VLIW embedded core jointly developed by Hewlett-Packard and STMicroelectronics [13][14].

The system (Figure 3) comprises a 'traditional' interpreted emulator for an SH3 ISA, complemented by enough system components (timers, serial ports, display, etc.) to be able to support a (simple) configuration of the WinCE kernel, after we wrote a hardware abstraction layer for it. The other main component of the emulation system is the SH to ST210 Just-In-Time compiler, which translates SH instructions into native ST210 instructions.

We designed the JIT to be transparent to the SH3 emulator, so that we can turn it on or off. The emulator invokes the JIT at every instruction fetch so that it can start, grow or stop translations for a new region of code, or release control to an already existing translation in the DELI code cache. At system bootstrap, the execution of emulated SH instructions takes place in the interpreted emulator. At the same time, the JIT produces new code fragments of translations and hands them to the DELI, occasionally releasing control to the few already translated fragments. After the initial warm-up phase, the DELI code cache is primed with the working set and most of the emulation time of the SH instructions is spent in the cache.

Coupling the JIT with an interpreted emulator is not mandatory (we could build a system based on the JIT alone); however, it allows the JIT to make better decisions

when producing the translations. For example, the JIT can grow code regions past branches if it knows the branch outcomes at JIT time; conversely, it would be hard to exploit this information with a translate-and-execute approach. The presence of the interpreted emulator also enables the JIT to 'bail out', so that it can make aggressive assumptions when generating the translation and then fall through to emulation for the few cases that fail.

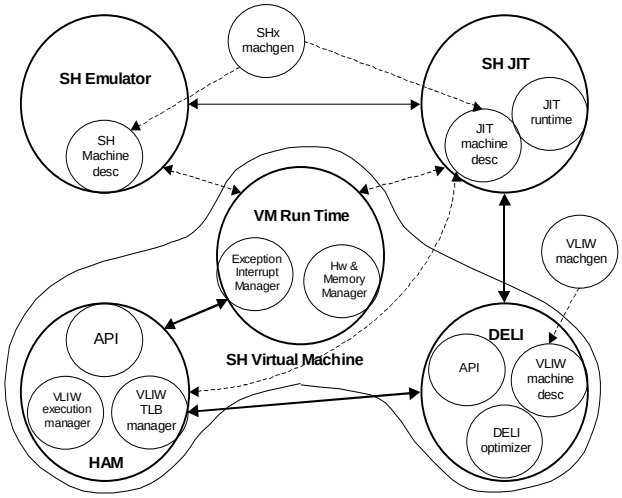

Figure 3 - Structure of DELIverX. It adds three components to the DELI: interpreted emulator, JIT compiler and Virtual Machine (VM) run time. The VM run-time, the interpreted emulator and the JIT depend on the target being emulated. The VM run-time leverages services from HAM to deliver a faithfully emulated machine context on exceptions/interrupts, at the same time providing a compatible VM system. The interpreted emulator and the JIT base their operations on the assumptions made by the VM run-time (the dotted arrows). The JIT also relies on some of the HAM capabilities to simplify the translations emitted through the DELI interface.

The JIT generates code for a run-time environment around which we built the SH virtual machine. In this world, the most difficult task is to efficiently handle a faithful emulation of the Virtual Memory (VM) system mapped onto the native VM. In general, the larger the distance between the VM parameters of the two systems, the greater the difficulties associated with it. In this particular case, we designed the Memory Management Unit (MMU) of the VLIW with a large degree of flexibility, exactly with the goal of simplifying the emulation task.

4.2. Performance Analysis

In this section, we report some experimental results to assess the performance of the DELIverX system. As we mentioned in section 4.1, our emulated processor is a Hitachi SH3 [19], and our target is a 4-wide VLIW embedded processor. Table 5 lists the most significant parameters of the two systems.

The ST210 does not currently include MMU support, so for our experiments we use a cycle-accurate instruction set simulator (ISS, validated against the hardware) augmented with virtual memory support, assuming a TLB model as described in Table 5. The ISS models the CPU core and the memory system, including caches, bus transactions and main memory timing. For the SH3, we use an *HP Jornada 548* palmtop running *PocketPC* and we measure wall-clock time average on multiple runs.

Table 5 - Parameters of the emulated and native systems

	Emulated Processor	**Native Processor**
CPU	Hitachi SuperH SH3, 133 MHz	Lx/ST210, 4-issue VLIW, 250MHz
L1 Cache	Unified 8KB 4-way I$+D$	32KB direct mapped I$, 32 KB 4-way D$
MMU	4-entry fully associative I-TLB; 32-entry, 4-way unified L2	32-entry, 2-way I-TLB; 128-entry, 16-way unified L2
Memory	32MB of DRAM	32MB of DRAM

Note that the ST210 has significantly higher performance than the SH3. While a comparison of the two might seem unfair, we have to keep in mind that supporting legacy code is a burden that is likely to slow down the processor evolution curve. To preserve binary compatibility in hardware, we have to resort to the adoption of a superscalar microarchitecture, which has severe consequences on complexity, cost and power efficiency. For example, while the SH4 (SH3 successor) is indeed a 2-way in-order superscalar, the designers of the SH5 (SH4 successor) decided against widening ILP, in favor of micro-SIMD extensions (a binary incompatible feature).

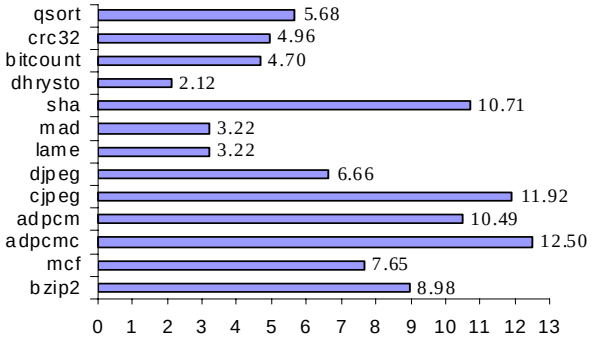

Figure 4 - Performance of the ST210 vs. the SH3 (SH3 = 1.0). The average is 7.02. The accumulated runtime of all benchmarks on the SH3 is about 65s (5s average); *qsort* is the shortest program (0.71s) and *lame* the longest (15.6s).

The benchmark suite is a combination of micro-kernels and real applications taken from the *MiBench* suite [18], *dhrystone* and a few of the SPEC-2000 (*bzip2* and *mcf*). We include audio (*adpcm*, *lame*, *mad*), and imaging (*jpeg)*

algorithms to approximate a representative workload for a next generation mobile computing appliance. We compile each benchmark for WinCE using the *Microsoft Embedded Visual C++* compiler (using the "maximize speed" configuration), and we run it in isolation within the PocketPC environment. DELIverX emulates exactly the same system, including the PocketPC environment. The standalone SH3 simulator (no JIT or DELI) on a Pentium III/800 runs 20x—30x times slower than the SH3.

To understand the raw speed difference between the SH3 and the ST210, Figure 4 shows that—on average—the ST210 is about 7 times faster than the SH3. Note that the ST210 runs without OS overhead, so this is an upper bound of what a real ST210-system could reach.

In Figure 5, we show the results of DELIverX running on the ST210 emulating the SH3. As we can see, we often achieve near-native SH3 performance (the average is 99% of native performance), and in some cases, we even exceed it (SHA is 83% faster emulated than on the SH3). The chart also shows the substantial benefits that DELI's dynamic code optimizer achieves (on average, a 61% improvement over non-optimized code).

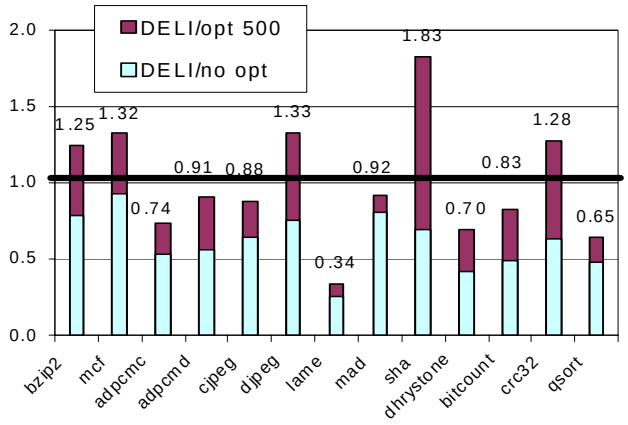

Figure 5 – DELIverX performance. The *y*-axis shows relative speedup (>1.0) or slowdown (<1.0) factors vs. the SH3 (y=1.0). The bars represent emulated code on the ST210 VLIW without optimization (*DELI/no opt*) and with optimization triggered at 500 executions (*DELI/opt 500*). The average is 0.62 without optimization and 0.99 with optimization.

Mixing native and emulated code

The last set of evaluations that we present deals with mixing native and emulated code. We consider the case where we introduce a natively compiled function within a fully emulated platform. In this example, we use DELIverX running PocketPC to invoke a cryptography kernel that executes the Secure Hashing Algorithm (SHA) to produce a key to be used for a secure transaction. SHA is a good example of code that benefits from instruction-level parallelism, and—as such—is a good target for a VLIW engine. In Figure 6 we can see the results of the experiment: by mixing native and emulated code we can get to about 50% of the speed of a native-only VLIW execution, and still more than five times faster than running the code on the SH3. While improvements in SH processor design are likely to close this gap, the advantages of the DELI approach are obvious in this particular example. In a more complex application, the gain would obviously depend on the relative importance of the natively compiled code.

This example illustrates one of the major benefits of using DELI: in a practical environment it enables developers to scale performance by selectively porting the compute-intensive kernels of applications, without giving up the benefits of a legacy operating system and GUI. In a rich-media world, performance is likely to be dominated by kernels (like SHA) that we can easily encapsulate to get the benefits of a VLIW engine. In this environment, the effect of a modest slowdown for GUI emulation is likely to disappear in light of the performance improvements in the computational kernels.

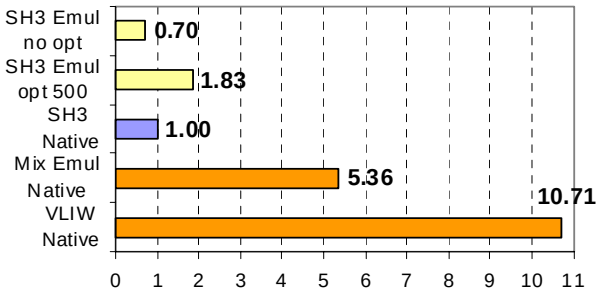

Figure 6 - Example of mixing native and emulated code for SHA cryptography (running for 4.18s). The values on the *x*-axis represent speedup factors vs. the base case of a native SH3 execution (SH3 = 1.0). The "*VLIW native*" case runs code in isolation (without OS); all others include the PocketPC overhead. The "Mix" case runs emulated PocketPC, with the SHA routine itself compiled natively for the ST210.

5. Summary and Conclusions

The DELI is a new run-time control point that lets its clients manipulate unmodified binaries in novel ways. It also allows us to build efficient emulators more easily and greatly enhance their functionality. The DELI unifies many techniques that make persistent changes at run time, but is not itself an emulator. Rather, it is a way of giving clients (including emulators) access to a caching and linking mechanism that operates at the lowest software level.

This paper describes the DELI, how it works, and what its interface looks like. It also presents some of the clients that DELI could support, including code manipulation, observation and emulation. The DELIverX Emulation prototype demonstrates the feasibility of DELI, and at the same time it helps us better understand the details of the DELI programming interface. DELIverX often achieves native performance, and the dynamic code optimizer adds substantial benefits (on average, a 61% improvement). Finally, we show that we can transparently mix native and

emulated code, yielding tremendous performance increases, without having to port the entire system, nor deal with a clumsy accelerator interface. We find these results impressive enough to report on today, but we believe we have a lot of headroom, and will achieve much better results as we continue our development.

The DELI is a new facility. While it would be easy to misunderstand it to be *yet another caching-and-linking rewriting system*, it is not that at all. It is an interface to the underlying mechanisms of such systems, exported for client use. Given the idea of exporting such an interface, one could do many new things. We have done a significant amount of development on it, have learned a lot, have convinced ourselves that it has many practical applications, and have been able to demonstrate some of its vast power. That said, we believe we have only scratched the surface of what can be done with it. At HP Labs Cambridge, we are continuing to build and refine the DELI infrastructure for different platforms and clients, the DELIverX prototype, and we are continuing to understand new ways in which we can use this new control layer.

Acknowledgements

Vasanth Bala was a significant participant in the original formulation of DELI.

References

[1] V. Bala, E. Duesterwald, S. Banerjia. Dynamo: a transparent dynamic optimization system. In SIGPLAN Conference on Prog. Lang. Design and Implementation, p. 1-12, 2000.

[2] T. Ball and J.R. Larus. Efficient path profiling. In Proc. of the 29th Int. Symp. on Microarchitecture, Paris 1996.

[3] T.Baji, N. Kawashimo, I. Kawasaki, and K. Noguchi. SuperH and SuperH-DSP Microprocessors for the Mobile Computing Age. Hitachi Review, Vol. 46 No.1. Feb 1997.

[4] B. Buck, and J.K. Hollingsworth. An API for runtime code patching. The International Journal of High Performance-Computing Applications, Vol. 14, no. 4, 2000, pp. 317-329.

[5] F. Chang, A. Itzkovitz, and V. Karamcheti. User-level Resource-Constrained Sandboxing. 4th USENIX Windows Systems Symposium, August 2000.

[6] W. Chen, S. Lerner, R. Chaiken, and D. Gillies. Mojo: A dynamic optimization system. In Proc. 3rd Workshop on Feedback-Directed and Dynamic Optimization, Dec. 2000.

[7] R.F. Cmelik and D. Keppel. Shade: a fast instruction set simulator for execution profiling. TR UWCSE-93-06-06, Dept. Comp. Science and Eng., Univ. Washington. 1993

[8] D. Deaver, R. Gorton, and N. Rubin. Wiggins/Redstone: An on-line program specializer. In Hot Chips 11, Palo Alto, CA, Aug. 1999.

[9] D. Ditzel. Transmeta's Crusoe: Cool chips for mobile computing. In Hot Chips 12: Stanford University. Aug. 2000.

[10] "*DynamoRIO*" http://www.cag.lcs.mit.edu/dynamorio/

[11] K. Ebcioglu and E. Altman. DAISY: Dynamic compilation for 100% architectural compatibility. In Proc. of the 24th Int. Symp. on Computer Architecture. Pages 26-37, 1997.

[12] K. Ebcioglu, E.R. Altman, E. Hokenek. A JAVA ILP Machine Based on Fast Dynamic Compilation. IEEE MASCOTS International Workshop on Security and Efficiency Aspects of Java. Eilat, Israel, January 9-10, 1997.

[13] P. Faraboschi, G. Brown., J. Fisher., G. Desoli, F. Homewood. Lx: A Technology Platform for Customizable VLIW Embedded Processing. Proc. 27th International Symposium on Computer Architecture (ISCA27). Vancouver, June 2000

[14] P. Faraboschi, F. Homewood, ST200: A VLIW Architecture for Media-Oriented Applications, Microprocessor Forum 2000, October 9-13 2000, San Jose, CA

[15] J. Fisher. Trace scheduling: a technique for global microcode compaction. IEEE Trans. on Computers, vol. 30, no. 9, 1981, pp. 478-490.

[16] J. Fisher. Walk-time techniques: Catalyst for architectural change. Computer, 30(9):40--42, September 1997.

[17] D.H. Friendly, S.J. Patel, Y.N. Patt. Putting the fill unit to work: dynamnic optimizations for trace cache microprocessors. In Proc. of the 31st Symp. on Microarchitecture, Dallas, 1998.

[18] M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge and R.B. Brown. *MiBench*: A Free, Commercially Representative Embedded Benchmark Suite. 4th Workshop on Workload Characterization, Dec. 2001, Austin, TX

[19] H. Maejima, M. Kainaga, K. Uchiyama. Design and architecture for low-power/high-speed RISC microprocessor: SuperH. IEICE Trans. on Electronics. Vol. E80-C, No.12. Dec. 1997.

[20] A. Klaiber. The Technology Behind Crusoe Processors. © 2000 Transmeta Corp. Available as: www.transmeta.com/pdf/white_papers/paper_aklaiber_19jan00.pdf

[21] T. Lindholm, Frank Yellin. The Java Virtual Machine Specification, Second Edition.

[22] C.M Merten, A. Trick, C.N. George, J.C. Gyllenhaal, and W.-M. Hwu. A hardware-driven profiling scheme for identifying program hot spots to support runtime optimization. In Proc. of the 26th Int. Symp. on Computer Architecture. Atlanta, Georgia. 1999.

[23] Microsoft Corp. .NET specifications. Available at http://www.microsoft.com/net/

[24] A. Robinson. Why Dynamic Translation? © 2001 Transitive Technologies. Available as: http://www.transitives.com/downloads/Why Dynamic Translation1.pdf

[25] E. Rotenberg, S. Bennett, and J.E. Smith. Trace cache: a low latency approach to high bandwidth instruction fetching. In Proc. of the 29th Int. Symp. on Microarchitecture, Paris, 1996.

[26] K. Scott, J. Davidson. Strata: a software dynamic translation infrastructure. Proc. Workshop on Binary Translation, 2000.

[27] A. Srivastava H. Edwards, H. Vo. Vulcan: Binary translation in a distributed environment. Technical Report MSR-TR-2001-50, Microsoft Research, 2001.

[28] D. Ung, C. Cifuentes. Machine-adaptable dynamic binary translation. Proc. ACM Workshop on Dynamic Optimization, ACM SIGPLAN Notices, vol 35, n.7, 2000, pp. 41-51.

[29] E. Witchel, and M. Rosenblum, "Embra: fast and flexible machine simulation", Proc. ACM SIGMETRICS Int. Conf. on Measurement and Modeling of Computer Systems, 1996, pp. 68-79

[30] A. Wolfe and A. Chanin, Executing Compressed Programs on an Embedded RISC Architecture, Proc. 25th Int. Symp. on Microarchitecture, pp. 81-91, Portland, OR, Dec. 1992.