

Advances and Future Challenges in Binary Translation and Optimization

ERIK R. ALTMAN, KEMAL EBCIOĞLU, MICHAEL GSCHWIND, SENIOR MEMBER, IEEE, AND SUMEDH SATHAYE

Invited Paper

Binary translation and optimization have achieved a high profile in recent years with projects such as the IBM DAISY open-source project, Transmeta Crusoe, HP Dynamo, Java JIT compilers such as LaTTe, and many others. Binary translation has several potential attractions: Architecture can become a layer of software, which allows the implementation of complex legacy architecture(s) through simple hardware and the introduction of novel new architecture and microarchitecture concepts without forcing any software changes. Secondly, binary translation enables significant software optimizations of the kind that would push the complexity boundaries if done with hardware alone. While still in its early stages, could binary translation offer a new way to design processors, i.e., is it a disruptive technology, the term popularized by Prof. Clayton Christensen? This paper discusses this interesting question, examines some exciting future possibilities for binary translation, and then gives an overview of selected projects (DAISY, Crusoe, Dynamo, and LaTTe).

One future possibility for binary translation is the Virtual IT Shop. Companies such as Loudcloud currently provide computational resources as services over the Web. These services are typically implemented through large and secure server farms. If a variety of customers are to be supported, a variety of architectures (x86, PowerPC, Sparc, etc.) must be present in the farm. Of necessity, the number of machines from each architecture is statically determined at present, thus limiting utilization and increasing cost. Binary translation offers a possible solution for better utilization: architecture as a layer of software, and hence dynamic configuration of the number of machines from each architecture in such farms.

The Internet is radically changing the software landscape, and is fostering platform independence and interoperability, with paradigms such as XML, SOAP, and Java. Along the lines of software convergence, recent advances in binary JIT optimizations also present the future possibility of a convergence virtual machine (CVM). CVM is similar to the Java Virtual Machine (JVM) in that both seek to facilitate a write-once, run-anywhere model of software development. However, the JVM suffers from the drawback that existing C/C++ applications and existing operating systems do not run on it. CVM aims to address the remaining research

challenges in allowing the same standard OS and application object code to run on different hardware platforms, through state-of-the-art JIT compilation and virtual device emulation.

Keywords—Binary translation, compilers, dynamic optimization, ILP, Java, JIT, VLIW.

I. CHALLENGES AND OPPORTUNITIES OF BINARY TRANSLATION AND OPTIMIZATION

Dynamic binary translation is just-in-time (JIT) compilation from the binary code of one architecture to another. Dynamic optimization is run-time improvement of code. This process is illustrated in Fig. 1. For this paper, we will assume such translation and optimization is performed by software at run-time and saved for some period of time, so that the overhead of translation and optimization may be amortized over multiple executions. Thus, we exclude from dynamic binary translation and optimization: 1) the scheduling and renaming done in hardware by out-of-order superscalar processors and 2) the hardware cracking of complex operations into simpler ones in superscalar implementations of architectures such as x86 [1], PowerPC [2], and S/390 [3], [32].

Despite these restrictions, many projects fit our definitions of dynamic binary translation and optimization. Some of these include Transmeta **Crusoe** [4], IBM **DAISY** [5], HP **Dynamo** [6], Compaq **FX!32** [7], as well as the many Java JITs such as **LaTTe** [8]. There is a vast amount of dynamic compilation work and we are unable to cover all of it in the space of this paper. Aside from our bibliography, interested readers are referred to [9], which has links to a very large number of dynamic compilation projects.

BT¹ has several qualities that are potentially attractive both from a research and a commercial point of view.

¹Dynamic binary translation and optimization are often implemented together, but sometimes separately. For simplicity when arguments apply to both, we will use the term **BT**.

Manuscript received January 5, 2001; revised June 15, 2001.

The authors are with the IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598 USA (e-mail: erik@watson.ibm.com; kemal@watson.ibm.com; mikeg@watson.ibm.com; sathaye@watson.ibm.com).

Publisher Item Identifier S 0018-9219(01)09690-6.

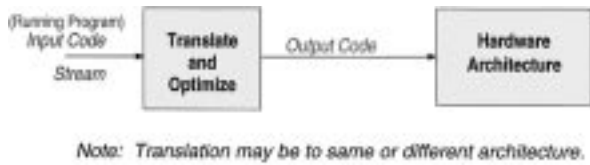


Fig. 1. Dynamic binary translation and optimization.

Improved Performance: **BT** is performed automatically at run-time, and can include optimization and profile-directed feedback without programmer intervention.

Architecture as a Layer of Software: The underlying machine may be unable to run legacy software directly, but the software layer fixes these problems. Thus, novel architectural ideas, that are seemingly incompatible with mainline legacy architectures, such as *x86* or *PowerPC*, can be used.

Reduced Hardware Complexity: Can save power since memory uses less power than logic. Unlike superscalar processors, **BT** translates and optimizes once and saves the result in memory. There is no need to redo the optimization (with logic transistors) each time the code is executed.

Given these advantages, one is tempted to ask a question related to a concept that has recently received a great deal of attention in corporations engaged in information technology (IT): Is **BT** a *disruptive technology* [10], [11], i.e., a technology which may start behind an established technology (such as out-of-order superscalars), but which improves more quickly over time, thus allowing it to overtake established technologies? Can **BT** change the way microprocessors are designed?

Unfortunately, disruptive technologies are best identified in perfect hindsight, such as in the case of 8-in disk drive technology overtaking the 14-in disk drives in the 1970s and 1980s. We do not have data now to conclude whether **BT** is disruptive. We are also aware of remaining technical difficulties in this research field (to be described later in this paper). But we can nevertheless identify some combinations of conditions where **BT** could be disruptive, and thereby provide a good context for the discussion of the following forward-looking **BT** research challenges, which we will offer to the reader in the hope of communicating our excitement about the future of **BT** research, before we delve into the analysis of **BT** systems available today.

Consider a simple **BT** microprocessor core and associated code implementing a very popular architecture as a layer of software, adopted as an open, nonproprietary standard in the future. Such a **BT** core and software can have some interesting IT implications, bordering on the disruptive.

- The core processor could become even more commoditized than *x86* is today.
- The rate of performance increase or power reduction could be higher in the simple standard core as compared to traditional microprocessors—due to more efficient focus of design team resources and hardware complexity management.
- Because of high hardware complexity, traditional microarchitectural enhancements tend to take a myopic view of code, aiming for a local optimum within a

narrow window. Dynamic software optimizations can afford a much more aggressive stance, and can allow significant global transformations.

Another feature which could strengthen the potential for **BT** to be disruptive is *commonality*. With different architectures implemented as different layers of software on the same core, an appropriately designed binary translation processor could efficiently emulate multiple “important” traditional architectures [12] (although not all architectures). Such a processor core (assuming the hardware simplicity is retained) would allow even more efficient utilization of the limited resources of hardware design teams. It could also vastly ease difficulties in migrating between architectures and could end the concept of captive IT customers—those for whom the difficulty and expense of moving to a new architecture is prohibitive.

An example of the potential beneficial application of commonality through **BT** is the *virtual IT shop* concept. According to an emerging vision of the future of IT among corporations (e.g., [13], [14]), computing hardware and software resources will be available over the Internet as if they were a utility, like the water supply of a city, allowing customers to increase or decrease their computing resources on demand (for example, HP has called this concept “e-services on tap” and “e-utilica” [15], [16]). As security technologies and Internet bandwidth improves, one could say that traditional in-house IT shops are likely to receive less emphasis over time, due to the economies of volume offered by the e-sourced computing model. The servers implementing the virtual IT shops will probably continue to be housed in large and secure server farms. However, the heterogeneity of architectures in the IT world today poses a problem. If some customers require *Sparc* engines, others need *x86*, and others need *S/390*, and yet other customers need a combination of these architectures in their virtual IT shop, how many does one stock from each architecture in the server farm to meet the customer needs? There is a resource allocation difficulty here.

A **BT** core with commonality can solve this problem by allowing virtual IT shops to maintain farms of identical processors, boards and racks with **BT** software, allowing the processors of the server farm to emulate the number and type of processors desired by customers. Individual processors in the farm could be configured to emulate a single processor at boot time, or could emulate multiple architectures in time-shared fashion through a hypervisor software layer. Either of these **BT** schemes can provide far better resource allocation and processor utilization than does a traditional server farm with fixed numbers of each processor type. They can also provide added reliability: When one virtual processor crashes due to the (customer) application software, the physical processor hosting it does not.

The Internet has recently been changing the software landscape in a radical way and has been implicitly encouraging *write-once, run-anywhere* software and interoperability of different hardware platforms, as exemplified by the recent popularity of technologies such as XML, Simple Object Access Protocol (SOAP) [17], and Java. Along the lines of *write-once, run-anywhere* software and interoperability, we

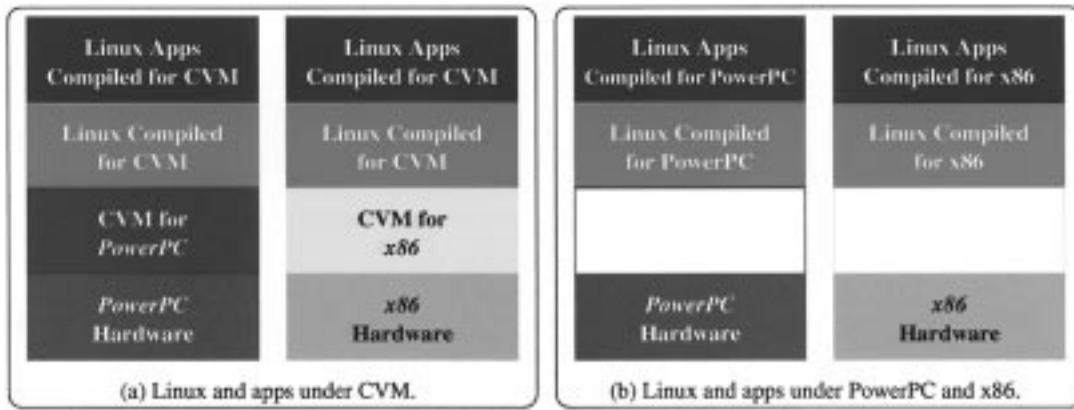


Fig. 2. Linux and applications with CVM and with today's heterogeneous platforms.

hereby propose yet another future research challenge for **BT** technology: it is what we term a *convergence virtual machine (CVM)*. CVM is similar to the *Java Virtual Machine (JVM)* and Java OS in the sense that both aim to facilitate a *write-once, run-anywhere* model for software development. JVM and Java OS, however, suffer from the drawback that existing C/C++ applications do not run in this environment, and neither do traditional operating systems such as Linux.

To remedy these problems, CVM makes a different set of tradeoffs than JVM. In order to provide security and safety guarantees, JVM in essence gives up being a universal machine, able to deal with all aspects of hardware and computation. CVM maintains universality at the cost of following the protection architecture of a typical modern RISC processor, which is less safe and secure than a JVM. Note, however, that a JVM can run on top of CVM.

CVM defines a simple but complete *convergence virtual machine* with RISC-like operations, standard virtual input–output (I/O) devices, and includes full virtual hardware details including interrupts, memory protection, and interprocessor communication. CVM could be implemented on top of multiple target architectures, and would use state-of-the-art JIT compilation and run-time emulation of the standard virtual I/O devices on each of the target architectures (We describe some of these state-of-the-art techniques later). An example of such CVM systems is given in Fig. 2(a), in which a common Linux OS binary and a set of Linux application binaries run on *PowerPC* and *x86*. By contrast, Fig. 2(b) shows the typical scheme today, where the Linux binary and applications are different on *PowerPC* and *x86*, and hence require software developers to test their code on multiple systems to ensure correct and robust performance. To take the CVM idea a step further, with CVM, object code could be in XML format, and operating systems could be booted from the web!²

With these forward-looking **BT** research challenges in mind, we now turn to the actual technical details of **BT**

²Architecture independent computing has been a longtime goal of computer science [18], and some earlier related efforts have demonstrated the partial feasibility of the *run-anywhere* object code idea (e.g., [19]). But the relative maturity and performance of the state-of-the-art dynamic compilation and optimization techniques today provide exciting research opportunities that we feel are likely to make a difference.

with some additional background in Section II. This is followed by overviews of four leading **BT** projects: IBM's **DAISY** in Section III, Transmeta's **Crusoe** in Section IV, HP's **Dynamo** in Section V, and Seoul National University and IBM's **LaTTe** in Section VI. Section VII provides a discussion and conclusions.

II. BACKGROUND

Binary Translation is a fairly large and complex field. Our intention in the rest of the present paper is only to give a summary overview of **BT**.

In this section, we will begin with defining a little **BT** nomenclature. We then put this nomenclature to use in discussing tradeoffs between different **BT** approaches and some of the hard problems faced in **BT**. We will then close by listing a few additional **BT** strengths, as well as some **BT** weaknesses.

A. BT Tradeoffs and Hard Problems

Several terms are frequently used in **BT**, whose definitions are provided as follows.

Source Architecture refers to the (legacy) architecture from which translation occurs.

Target Architecture refers to the architecture to which translation occurs.

Virtual Machine Monitor (VMM) is the software controlling binary translation. The **VMM** provides an abstraction of a machine, e.g., *PowerPC*, *x86*, *PA-RISC*, or *JVM*. Transmeta uses the term **CMS** (Code Morphing Software) for **VMM**.

Translation Cache (TCache) is the memory where translations are stored. The **TCache** is not necessarily a hardware cache.

BT can be performed in myriad ways for myriad cases. The systems—**DAISY**, **Crusoe**, **Dynamo**, and **LaTTe**—that we look at in detail in Sections III–VI cover several of the following tradeoffs:

- interpreting versus translating/optimizing;
- emulating a **virtual** machine versus emulating a **real** machine;
- full system versus user mode only;

- OS independent versus OS dependent;
- translating to a different architecture versus translating to the same architecture;
- emulating a single source architecture versus emulating multiple source architectures.

Even with so many tradeoffs, there are a number of difficult issues that must be dealt with by nearly all **BT** systems.

Self Modifying Code: This issue is not limited to old or esoteric user code. Full system **BT** must deal with programs being loaded and unloaded by the operating system. An old program overwritten by a newly loaded one presents the same problem of self modifying code as programs that intentionally modify themselves. In both cases, when code in the *source architecture* is changed, any translations of that code must be invalidated.

Precise Exceptions: Both synchronous exceptions such as page faults and asynchronous exceptions such as timer interrupts must present a proper and consistent view of the *source architecture* state to the (translation of) the *source architecture* exception handler.

Address Translation: Full system **BT** must deal with the fact that virtual addresses computed by instructions do not indicate the underlying physical location holding the desired value, and that the physical location may not even exist (page fault condition). Further complications are possible in that certain addresses may be noncacheable, for example, in the case of memory mapped I/O.

Self Referential Code: To ensure that they are not corrupted or for other reasons, some programs checksum themselves or otherwise look at their own code. Hence, a **BT** system must ensure that such self-referential behavior works properly. This is generally accomplished by leaving a copy of the *source architecture* code in its original address range for the translated version (which is stored in another region of memory) to look at.

Management of Translations: The size of the **TCache** is limited (and ranges in size from a few hundred thousand bytes to a few hundred million bytes in the **BT** systems examined here). When the **TCache** becomes full, room must be made for new translations. Likewise if old translations are invalidated (e.g., by self-modifying code), the **TCache** should reflect this fact. **BT** approaches range from flushing the entire **TCache** when it becomes full or when a change is anticipated in workload behavior (**Dynamo**) to using a form of generational garbage collection (**DAISY**).

Real-Time Behavior: Some code in some systems has a time limit on when it must complete or a rate at which it must execute. These limits can be problematic for **BT** systems, which generally have considerable variation on execution speed depending on whether a particular fragment of code has been translated. In some (but not all) cases, real-time code can be determined ahead of time (heuristically) and pre-translated.

Boot and BIOS Code: In full system **BT**, the lowest level code controlling the *source architecture* must be faithfully translated for the *target architecture*. This may require some knowledge of the *system source architecture* as well as the *processor source architecture*. In developing **DAISY**, for ex-

ample, we noticed that *PowerPC source architecture* boot code disabled DRAM banks during memory testing. Since the **DAISY VMM** resided in those DRAM banks, faithful execution of this boot code killed **DAISY**. Thus, not only is it important for **DAISY** to emulate the *PowerPC* architecture, **DAISY** should emulate the memory controller (and prevent disabling **DAISY** DRAM) as well. Transmeta **Crusoe** also includes some memory controller functions.

Reliability and Correctness: To gain user trust and acceptance, **BT VMM** software must be extremely robust, and at least as free of bugs as the underlying hardware. However, this difficult problem is mitigated slightly by the fact that **VMM** bugs should be easier and cheaper to fix than those in hardware.

Code Reuse: Since **BT** uses software to translate code from the *source architecture* to the *target architecture*, translation takes considerable time. In **DAISY**, for example, about 4000 operations are needed to translate and optimize a single *PowerPC source architecture* instruction. To amortize this translation and other **VMM** overhead, the resulting *target architecture* translation must be reused many times. Reuse rates have several other effects.

- High code reuse permits time-consuming high optimization and ILP extraction.
- Conversely, high overhead translators and interpreters require high code reuse to amortize the time they take. Optimizations that are profitable in a static compiler can take too long to amortize in a dynamic one.
- Large *TCaches* allow more reuse at the cost of memory and lower adaptability to code changes.

We shall discuss several of these issues in more detail later in the sections on **DAISY**, **Crusoe**, **Dynamo**, and **LaTTe**.

B. *BT* Strengths and Weaknesses

Well-designed **target architectures** and **VMMs** can generally handle the problems outlined in Section II-A, although hard real-time requirements can still present problems, as can very low reuse rates. **BT** has additional drawbacks as well. 1) **BT** and the **VMM** take away cycles from *source architecture* programs. 2) **BT** and the **VMM** take memory and resources from the *source architecture* machine. 3) **BT** can be slow, particularly when a program starts executing, since all code must initially be interpreted and translated to *target architecture* code. 4) Debugging the **VMM** can be difficult: *Target architecture* code is several times removed from source code. Behavior can be nondeterministic in real systems. 5) **BT** is brand new design territory, and finding a sweet spot can be difficult. Even the best engineers can get stuck in suboptimal areas.

Luckily, in addition to the **BT** advantages listed in Section I, there are numerous additional positive points to offset these drawbacks.

- Legacy binaries where source code is unavailable can be optimized.
- Many commercial programs are compiled with debug support enabled during development. Time constraints on shipping the program often do not allow time for ad-

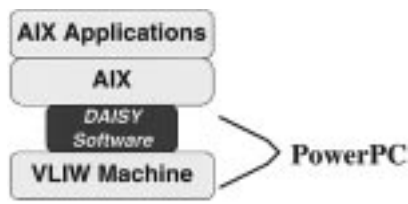


Fig. 3. DAISY schematic.

ditional testing (and bug fixes) with compiler optimization enabled, meaning that no traditional optimizations are done for many commercial programs. With appropriate architectural support, **BT** permits such optimizations under the covers when the user runs the program.

- **BT** is not limited in optimization scope and can cross boundaries such as *indirect calls*, *function returns*, *shared libraries*, and *system calls*.
- Translated basic blocks can be laid out contiguously in the order they are naturally and most frequently visited. This helps instruction cache performance.
- Future architectural improvements are transparent to the user. In this vein, **BT** allows compatibility between VLIWs of different sizes and generations—the distribution format is that of the *source architecture*, with code being translated by the **VMM** to the *target architecture* VLIW. Transmeta already claims to have made use of this advantage with different versions of **Crusoe**.
- If better compiler algorithms are found, only a software patch is needed to install them and update the **VMM**.
- If a bug is found in the **VMM**, a software patch is sufficient to fix it.
- Some hardware bugs such as nonworking opcodes may be worked around by changing the **VMM** software.
- With **BT**, intelligence is in software, not hardware. This tends to result in smaller chips with commensurately higher yield.

III. IBM DAISY

Most **DAISY** work uses *PowerPC* as the *source architecture*, as depicted in Fig. 3, where *DAISY Software* (i.e., the **DAISY VMM**) and the **DAISY VLIW** architecture together implement the *PowerPC* architecture. However, the **DAISY** approach is general: in [12], we describe how to support multiple architectures with the **DAISY** approach, and describe in more detail how we envision to use **IBM S/390** as the *source architecture*. In addition, the 1996 **DAISY** Research Report [5] discussed **PowerPC**, **S/390**, and **x86** as *source architectures*.

Fig. 4 is a block diagram for a **DAISY** system. In this system, the **DAISY VMM** code is stored in the **DAISY FLASH ROM**. When the system powers up, the **VMM** code is copied to the **DAISY** portion of memory, and execution begins. After the **VMM** initializes itself and the system, it begins *PowerPC* emulation by translating the *PowerPC* firmware in the **POWERPC FLASH ROM** and executing it on the **DAISY VLIW**. Then, the translated firmware loads the operating system (**AIX**), which **DAISY** likewise translates and executes. Eventually the system is ready to use and

user applications are translated and executed as needed. All translated code is kept in a memory region reserved for **DAISY** software (and may also reside in instruction caches not shown in Fig. 4).

DAISY's **VMM** and translator have been implemented and are sufficiently robust that they have booted an **RS/6000** workstation under simulation. This boot process under **BT** includes translation and execution of firmware, the **AIX** operating system, **X-Windows**, and numerous **AIX** utilities and other programs. Naturally, the **DAISY VMM** translates *PowerPC* code to **VLIW** code in this “**BT** boot.” The simulator then translates the **VLIW** code back to *PowerPC* code emulating **DAISY**'s **VLIW** instructions. The simulator also performs tasks such as address translation that would be done by hardware in a real **DAISY** system.

Fig. 5 provides more detail about how **DAISY** translates and executes code. **DAISY** begins by interpreting *PowerPC* instructions in the box at the upper left of Fig. 5 and adding them to a group **X**. After interpreting each instruction, **DAISY** checks whether the instruction would make a good *stopping point*. As shown in Fig. 6, good stopping points are instructions such as function returns and loop back branches encountered after a certain number of instructions (e.g., 24 instructions) have been seen, or sufficient parallelism (e.g., 3 IPC) has been found in the *PowerPC* instructions of group **X**. The desired parallelism and number of instructions vary depending on whether the instructions in group **X** have been executed frequently or not.

Returning to **DAISY**'s operation in Fig. 5, if an instruction is not a good stopping point, **DAISY** interprets the next *PowerPC* instruction. Eventually a good stopping point is found, at which time **DAISY** terminates group **X** renames it **Y**, and starts a new group **X**. If group **Y** has been executed 30 times in this manner, then **DAISY** translates it to **VLIW** instructions for speedy execution next time **Y** is encountered. If group **Y** has *not* yet been executed 30 times, **DAISY** checks if the instruction after **Y** constitutes the start of a previously translated group. If not, it continues interpretation in the box at the upper left of Fig. 5. If the *PowerPC* instruction following group **Y** is the start of a previously translated group, that translated group is executed as shown in the box at lower left. After the translated group finishes execution, it checks if there is a translation starting at the *PowerPC* instruction following the group exit. If a group is found, control transfers to it and execution of the code occurs at native **VLIW** speed. If no translated group is available, the *PowerPC* code is interpreted and the next group is translated. (The check for a translated follow-on group is conceptual, as **DAISY** can branch directly from translated group to translated group. If a group is destroyed at some point, its predecessors must be updated so as not to branch to it anymore.)

When **DAISY** starts or new code is encountered, most of the time is spent in the two boxes at the top left of Fig. 5, i.e., interpreting and looking for a stopping point. However, when **DAISY** has been running for a while and most code has been translated to **VLIW** instructions, little interpretation is done, and most time is spent in the middle and lower left boxes in Fig. 5, i.e., almost all time is spent in translated **VLIW** code.

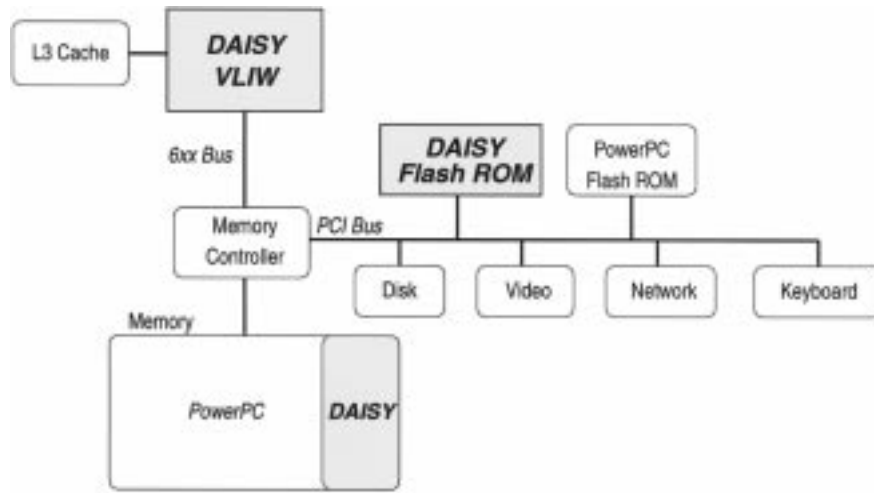


Fig. 4. DAISY system.

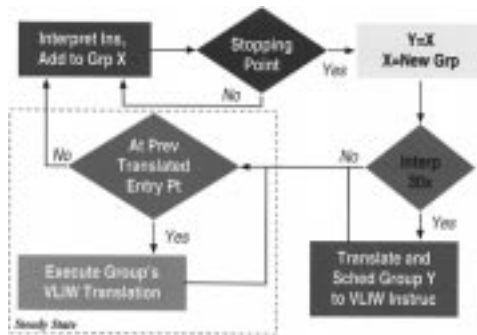


Fig. 5. DAISY translation and execution of translated code.

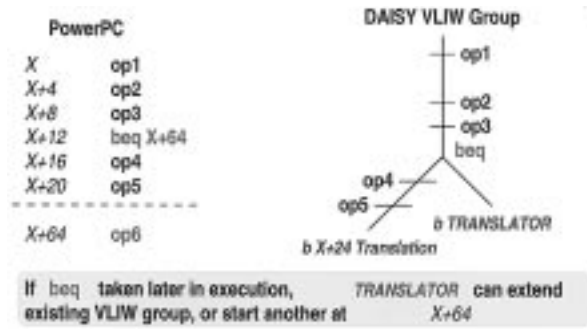


Fig. 7. PowerPC code and corresponding DAISY group.

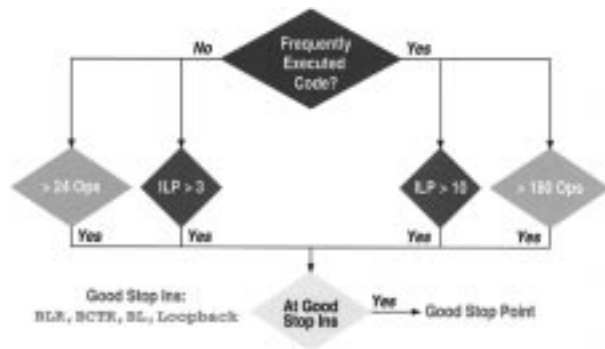


Fig. 6. DAISY stopping points.

As can be seen from Fig. 5, at a given time DAISY interprets and translates code along a single path into a group. However, if DAISY later finds that a particular exit of a group is taken frequently, DAISY schedules operations from beyond that exit into the group. In this way, groups are not limited to single paths, but can become trees. For example, in Fig. 7, when DAISY first sees the snippet of PowerPC code at left, the beq instruction normally falls through. Thus, DAISY produces the VLIW group shown at the right of Fig. 7 with operations from above the branch and from the fall-through path, i.e., with op1 to op5.

Later, however, DAISY may notice that the VLIW group in Fig. 7 frequently exits at the lower right via the b TRANS-

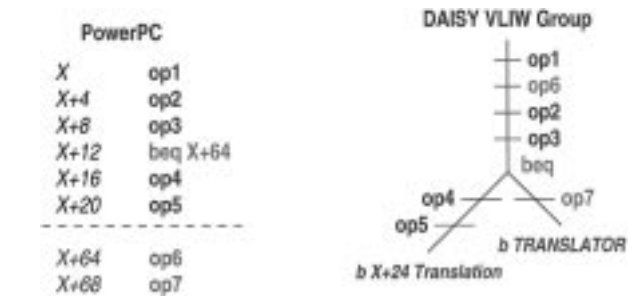


Fig. 8. Extending a DAISY group.

LATOR. Such frequent exits signal DAISY that this group should be extended with operations from this exit. Fig. 8 depicts the result of this extension. Op6 and op7 from the beq target have been added to the VLIW group at right, converting the straightline group into a tree. Op6 has been scheduled speculatively above the branch, which the DAISY scheduler does if there is an available function unit and all input values for op6 are available at that point. Op7 is scheduled on the right leg of the group. Apparently there were either no available function units earlier or its inputs were not ready.

By allowing operations from multiple paths in its groups, DAISY is not dependent on good branch prediction, and indeed makes forward progress along all possible paths. The tree form of DAISY groups comes at the cost of some code explosion via tail duplication. However, tree groups simplify

<pre> def r1 STORE r1, (X) ... LOAD r2, (X) ... STORE r2, (Y) ... LOAD r3, (Y) addi r4=r3,1 </pre> <p>(a) Original Code</p>	<pre> def r1 addi r63=r1,1 STORE r1, (X) ... LOAD r2, (X) ... STORE r2, (Y) ... LOAD r3, (Y) copy r4=r63 </pre> <p>(b) Use moved up through memory dependences.</p>
---	---

Fig. 9. **DAISY** and *load-store telescoping*.

some optimizations. For example, only one reaching definition is possible at a given point, and register allocation is simplified.

DAISY performs a variety of optimizations including *ILP scheduling with data and control speculation, loop unrolling, alias analysis, load-store telescoping, copy propagation, combining, unification, and limited dead code elimination* [5], [20]–[22]. Some of these enable **DAISY** to perform optimizations not possible in a superscalar processor. For example, *load-store telescoping* allows dependences through memory to be reduced to register dependences in some cases, as illustrated in Fig. 9. In Fig. 9(a), the bottom use of `r3` in `addi r4 = r3, 1` depends through a series of memory operations on the `def r1` at top. **DAISY** is able to recognize this dependence and move the `addi` operation before all the memory operations and immediately following its `def` as shown in Fig. 9(b). In performing *load-store telescoping*, **DAISY** knows that the desired input value for `addi` is no longer in register `r3`, but in `r1`. There are many cases like this where *load-store telescoping* allows dependence chains to be significantly shortened. Reference [22] provides more details of this optimization.

When moving operations like `addi`, **DAISY** renames any results produced such as `r4`. Renamed values go in **DAISY** scratchpad registers `r36` to `r63`. In this case, the `addi` result is placed in `r63` instead of `r4`. **DAISY** registers `r0` – `r31` coincide with *PowerPC* registers, and in order to maintain precise *PowerPC* exception semantics, **DAISY** puts result values into *PowerPC* registers in original program order. Thus, in Fig. 9(b), `r63` is copied to `r4` at the point at which `r4` was defined in Fig. 9(a).

Not only does **DAISY** speculate nonexception causing operations like `addi`, it also speculates loads. To do this safely, speculative loads are quashed by **DAISY** hardware if they attempt to access I/O space and an exception tag (poison bit) [23], [33] is set for the register in which the load result was to go. This exception tag is checked at the in-order location of the load, and if set, causes an exception to the **DAISY** **VMM**, which in turn reexecutes the load in order. Exceptions such as page faults are similarly deferred to their in-order location—after all, a speculative load may turn out not to have come from the actually executed path. Such hardware

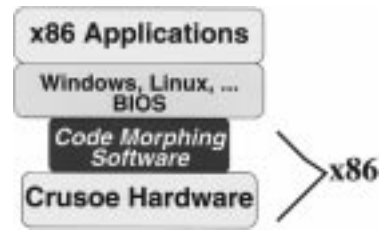


Fig. 10. **Crusoe** system.

support allows **DAISY** to perform speculation and optimization not possible in **BT** systems such as **Dynamo** running on unmodified conventional processors. (As described in Section IV, **Crusoe** also has such hardware support, although the details differ significantly from **DAISY**.)

Even with this hardware protection for misspeculated loads, there can be a problem if a load is speculated above an aliasing store. To handle this problem, **DAISY** uses a **load-verify** operation [24].

- A **load-verify** is inserted at the in-order location of the load.
- **Load-verify** reloads the value and checks if it is the same as the speculatively loaded value.
- If **Yes**, the speculation is okay—execution is continued normally.
- If **No**, the speculation is bad—the **DAISY** processor traps to the **VMM** and enters recovery code.
- Speculative values in non-**PowerPC** registers can be discarded. Execution can resume by using the in-order value of the load.
- If a particular **load-verify** fails frequently, the **VMM** can retranslate the block of code, so as not to speculate this load.
- There is a very limited time budget available at run-time for alias analysis. With the **load-verify** scheme, simple alias analysis suffices.

We close this section on **DAISY** by noting that a version of **DAISY** is now available as open source. Details, documentation, and code can be found at: <http://oss.software.ibm.com/developerworks/opensource/daisy>.

IV. TRANSMETA CRUSOE

The Transmeta **Crusoe** shares several elements with **DAISY**. Fig. 10 illustrates a **Crusoe** system, and can be compared to the **DAISY** system in Fig. 3. The significant difference is that **Crusoe** emulates an *x86* system, while **DAISY** emulates a *PowerPC* system. Both perform full system emulation including not only application code, but operating systems and other privileged code.

Although it may not be visible from these figures, there are several other similarities. Both use an underlying VLIW chip specifically designed to support **BT** for the source architecture (*x86* and *PowerPC*). Both also aim for high performance. In benchmark tests, **DAISY** can complete the equivalent of 3–4 *PowerPC* instructions per cycle. Transmeta has claimed that the performance of a 667-MHz **Crusoe** TM5400 is about the same as a 500-MHz Pentium III [25].

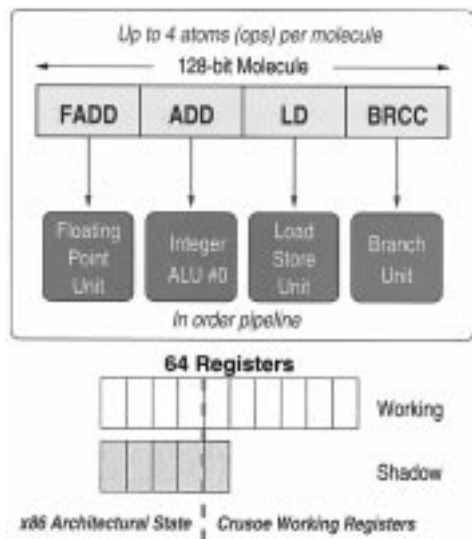


Fig. 11. Crusoe processor block diagram.

Aside from the different source architectures emulated, **Crusoe** and **DAISY** differ in their intended use. **DAISY** is designed for use in servers and consequently is a big machine capable of issuing 8–16 instructions per cycle, and with gigabytes of total memory. Given this large machine, the **DAISY VMM** emphasizes extraction of parallelism when translating from *PowerPC* code. **DAISY** also reserves 100 MB or more for itself and its translations.

Crusoe is aimed at low power and mobile applications such as laptops and palmtops, and is consequently smaller, issuing only 2–4 instructions per cycle and having 64–128 MB of total memory in a typical system. Thus, **Crusoe** reserves only 16 MB for itself and its translations.

Fig. 11 shows the core of a typical **Crusoe** processor. Like **DAISY**, **Crusoe** reserves some of its 64 registers for their *x86* equivalents, allowing the rest to be used for speculation and scratchpad uses. Unlike **DAISY** however, **Crusoe** maintains a *shadow* copy of the *x86* portion of its register set. When a translated group completes, the contents of all *working x86* registers are copied at once to the *shadow* register set. This *shadow* copy allows **Crusoe** to efficiently recover from exceptions. For example, if a page fault occurs in the middle of a translated group, **Crusoe** rolls back to the start of the group by restoring the values in the *shadow* registers to the *working* registers. It is then possible to step through the *x86* instructions one by one until the exception is found and handled.

Rolling back also requires that any stores performed since the beginning of the translated group be flushed. **Crusoe** provides hardware support for this in the form of a *gated store buffer* [26], which holds all stores as pending until a translated group successfully completes, and if the group does not successfully complete, quashes all stores arising from the aborted execution of that group.

This ability to back out of executing a group offers some optimization opportunities for the **Crusoe** translator which are not available to its **DAISY** counterpart. Within the scope of a group, instructions can be arbitrarily reordered or deleted. Thus, **Crusoe** can perform optimizations such as strength reduction and aggressive dead code elimination

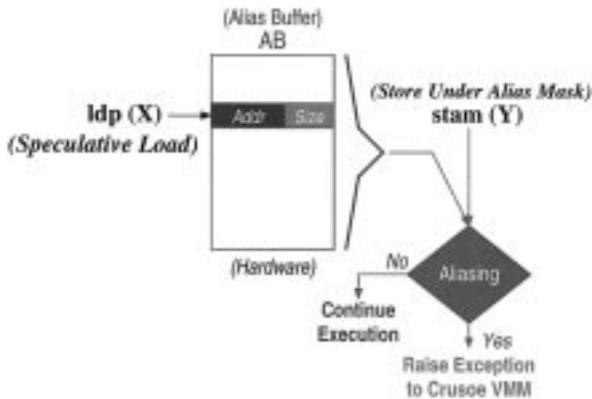


Fig. 12. Crusoe and speculative loads.

which are not currently possible in **DAISY**. However, by using a more sophisticated state recovery mechanism such as described in [27], [34], these optimizations can be performed even without specific hardware support.

As illustrated in Fig. 12, **Crusoe** also recovers from misspeculated loads differently than **DAISY** with its *load-verify* mechanism. **Crusoe** speculative loads are converted to *ldp*—*load and protect* operations [28], [29], [4]. *Ldp* operations put an entry in the *Alias Buffer (AB)* in **Crusoe**. Each entry contains the address loaded from and the number of bytes loaded. Stores with which *ldps* may alias are converted to *stam*—*store under alias mask* operations. As shown in Fig. 12, when *stam* operations execute, they check the *AB* for aliasing *ldps*. If none are found, execution continues normally. If aliasing is found, the **Crusoe** processor raises an exception and traps into the **Crusoe VMM** (or **CMS**) results. **CMS** then rolls back to the previous checkpoint, thereby discarding the execution sequence which violated the ordering constraints on memory operations and reexecutes code more conservatively.

This *ldp-stam* scheme offers two particular advantages over *load-verify*. 1) The number of operations executed is lower. Instead of loads being converted into a speculative load and a *load-verify*, they are only converted into a speculative load (*ldp*). 2) Consequently, the load-store bandwidth is also reduced. However, the *ldp-stam* scheme has the drawback that the number of speculated loads in a translated group is limited by the size of the *AB*. *Ldp-stam* also has the drawback that false aliasing may occur. For example, a speculative load from an untaken path may alias with a store from a taken path, resulting in an unneeded exception.

In other ways, **Crusoe** optimization is similar to that of **DAISY**. Code is first interpreted and profiled. If a fragment turns out to be frequently executed (more than 50 times), it is translated to native **Crusoe** instructions.

Fig. 13 depicts a block diagram of the TM5400 **Crusoe** chip. There are several interesting points to note about the TM5400.

- It has 8 kB of on-chip local memory for data and 8 kB of on-chip local memory for instructions. These presumably allow the most critical parts of **CMS** to remain resident for quick access and without perturbing *x86* code and data.

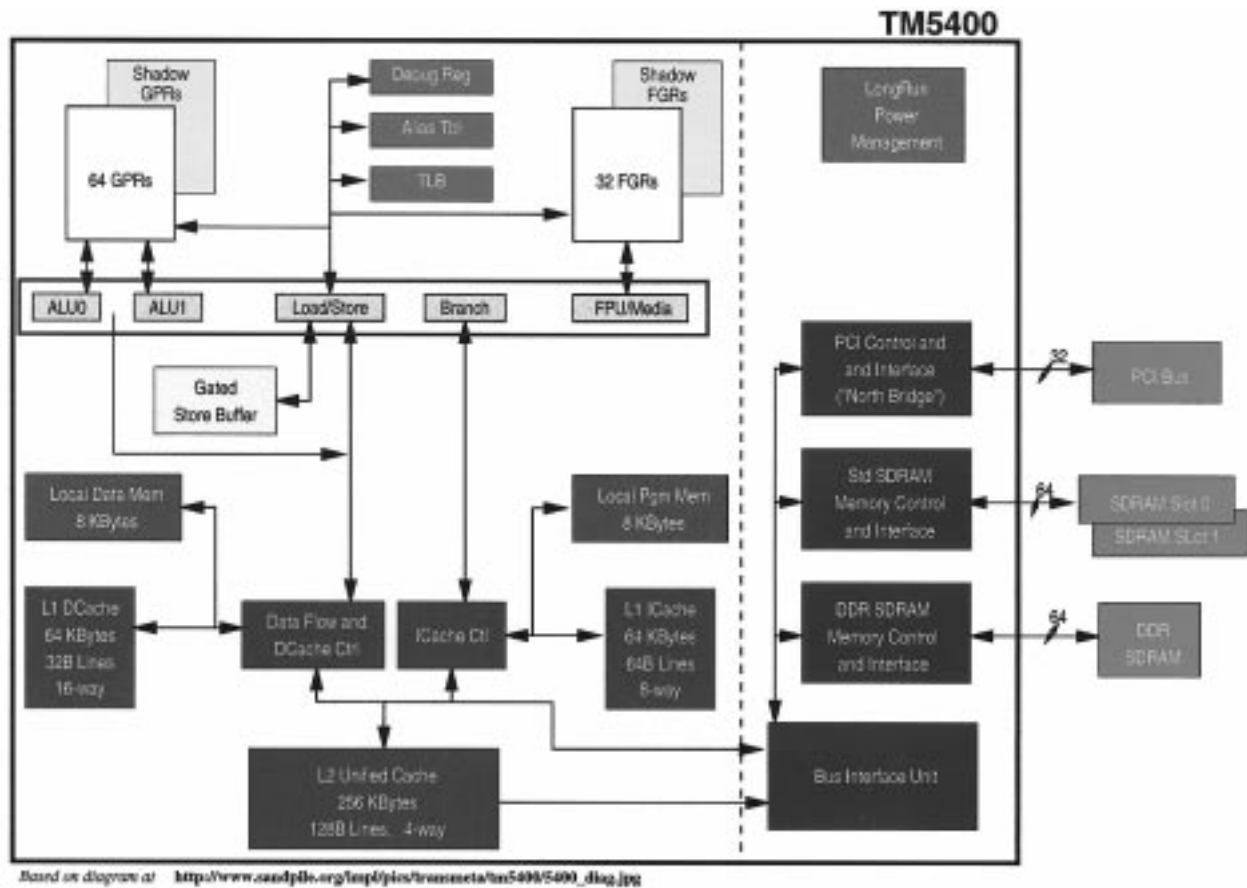


Fig. 13. Crusoe TM5400 processor.

- The high 16-way associativity of the L1 DCache acts to minimize conflicts between data used by the **x86** code and that used by CMS.
- As noted previously, the memory controller is integrated into the TM5400 because it is part of the standard PC architecture.
- The TM5400 has about 7 million transistors compared to larger numbers of transistors in mainstream AMD and Intel microprocessors, illustrating that **BT** allows substantial reduction in hardware complexity.

V. HP DYNAMO

The HP **Dynamo** has a slightly different structure from **DAISY** or **Crusoe**, as can be seen by comparing Fig. 14 to Figs. 3 and 10. Instead of running below the operating system like **DAISY** and **Crusoe**, **Dynamo** runs above the HP/UX operating system. While this offers some advantages for the software architecture of **Dynamo**, it also exposes **Dynamo** to the application and is hence less transparent than **DAISY** or **Crusoe**.

By running in the virtual address space of a single HP/UX process, **Dynamo** does not need to deal with address translation and translation groups spanning multiple pages. This reduces the number of synchronous exceptions which need to be handled correctly significantly, thereby allowing more aggressive code optimizations. While this simplifies the ar-

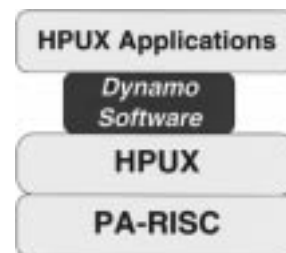


Fig. 14. Dynamo system.

chitecture, synchronous exceptions are not eliminated completely, as they are available through the Unix *signal* abstraction. To deal with this, **Dynamo** offers a conservative mode to applications which depend on the precise semantics of the Unix *signal* services. A more general way of implementing synchronous exceptions correctly in the presence of aggressive optimization has been described in [27], [34].

In addition, **Dynamo**'s translations last for only a single invocation of a program, whereas **DAISY** and **Crusoe** translations may last an arbitrarily long time and over many invocations of a program. Compaq's FX!32 exploits operating system services to create a persistent cache of translations on the disk. This allows translations to be used across multiple invocations of a program [7]. Access to operating system services and hence devices is one of the advantages offered by systems operating above the system level. Full-system translators such as **DAISY** and **Crusoe** are strictly architecture

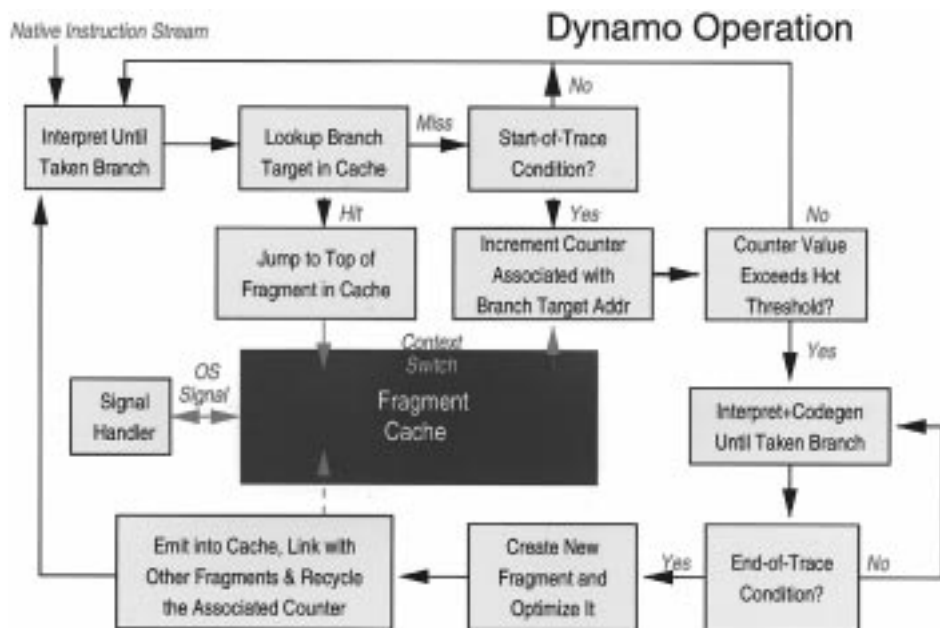


Fig. 15. Dynamo operation.

emulators, and consequently are unaware of disks, networks, and other system elements.

Perhaps the biggest difference between **Dynamo** and **DAISY/Crusoe** is that **Dynamo** does not really “translate.” **Dynamo** takes PA-RISC code as its input and produces further optimized PA-RISC code as its output. **Dynamo**’s stated goal is “. . . to use a piece of software to improve the performance of a native, statically optimized program binary, while it is executing” [6]. **Dynamo** succeeds in this goal, improving execution time on some **SPEC** benchmarks by up to 22% and by an average of more than 10%.

The fact that **Dynamo** does not actually translate between a source and a target architecture provides a benefit unavailable to **DAISY** or **Crusoe**. If **Dynamo** determines that its overhead is costing more than its optimization is helping for a particular program, it bails out and starts to execute the unmodified original program code directly.

Dynamo’s operation is detailed in Fig. 15, which can be compared to Fig. 5, which details **DAISY**’s operation. Overall **Dynamo** and **DAISY** operate in a similar manner, first interpreting *source architecture* operations, then translating them into groups for the *target architecture* if they are frequently executed. (As already noted, **Dynamo**’s *target architecture* is the same as its *source architecture*.) Like **DAISY**, **Dynamo** ends groups at good stopping points such as loop back branches. Also like **DAISY**, **Dynamo** groups can traverse indirect branches, function calls and returns, and virtual function calls. However, unlike **DAISY**, **Dynamo** groups are always a single path—they are never expanded to become trees or have other forms. This tends to limit code explosion at the cost of possible losses in exploiting parallelism.

Dynamo’s optimizer is efficient and operates with only a single forward and single backward pass. The optimizer eliminates unconditional branches since translated groups are placed contiguously in memory. To achieve this contiguity, **Dynamo** inverts the sense of conditional branches

if the taken path is the one in the group. **DAISY** and presumably **Crusoe** also perform this straightening which aids ICache and instruction fetch performance. However, **Dynamo** benefits particularly from this code straightening on the HP PA-8000, perhaps because of the less robust I-TLB hierarchy on this machine.

Among **Dynamo**’s other optimizations are *copy propagation*, *constant propagation*, *strength reduction*, *loop invariant code motion*, and *loop unrolling*. Like **DAISY**, **Dynamo** also branches directly between groups without invoking its **VMM** to determine if a successor translation exists and where it lies. This direct branching results in a dramatic 40× improvement in performance.

Dynamo follows a policy of flushing its entire TCACHE whenever it becomes full or when **Dynamo** anticipates a change in workload behavior. **Dynamo**’s flushing policy has other benefits: 1) it allows **Dynamo** translation groups to adapt to new code patterns and 2) it allows for a quick reset of many data structures and memory pools without costly garbage collection. However, **Dynamo** has a 500-kB TCACHE, which is relatively small compared to **DAISY**’s 100-MB or larger TCACHE, meaning that it takes **Dynamo** a relatively shorter time to recover from a full TCACHE flush than would **DAISY**.

VI. SEOUL NATIONAL UNIVERSITY—IBM LaTTe

LaTTe is one of many projects providing a JVM and JIT.³ **LaTTe** is fast and effective, with performance generally 8% to 26% better than commercially available products from Sun [30]. This good performance makes **LaTTe** an interesting implementation to examine. In addition, since **LaTTe** executes on Sparc systems, direct comparisons with Sun’s Java products are possible.

³**LaTTe** is a university collaboration between IBM and Seoul National University (SNU) begun in November 1997.

Unlike **DAISY**, **Crusoe**, and **Dynamo** which seek to use **BT** to improve the performance of real legacy machines, **LaTTe** seeks to use **BT** to improve the performance of a virtual machine, and in some sense reduce the semantic gap between the *source architecture* JVM and the underlying *target architecture*. **LaTTe** and *Java* have several advantages over **BT** for traditional architectures. There is no problem in discovering code in *Java* or in knowing higher level semantic information such what code constitutes a method. Self-modifying code, address translation, and BIOS code are likewise not present in *Java*.

Several elements contribute to **LaTTe**'s performance. **LaTTe** uses a lightweight monitor optimized for single threaded programs. **LaTTe** also has efficient exception handling. Instead of inserting code to check for and handle exceptions, **LaTTe** uses hardware generated signals such as segmentation violations to detect exceptions such as out-of-bounds memory accesses. Since exceptions are normally the *exceptional* case, this policy dramatically improves execution time in the normal case when exceptions are not thrown.

LaTTe also benefits from efficient garbage collection and memory management. Key to **LaTTe**'s efficient garbage collection is an algorithm which takes time proportional to the number of live items in the heap, as opposed to time proportional to the size of the entire heap as is the case for traditional mark and sweep algorithms [31].

LaTTe converts virtual method calls to direct method calls or inlines them by including a specific (conditional branch) check for the most frequently occurring method invoked from a particular call site. If the check passes, the frequently occurring method may be branched to directly. The **LaTTe** JIT compiler performs an additional optimization for calls to virtual methods that are *de facto* final. The current class hierarchy is analyzed while generating code. If a virtual method call can currently be resolved to only one method at the present call site, the JIT compiler optimistically generates code for a final method call (without any run-time checking), and inlines that method if appropriate. The call site is then backpatched to do a proper virtual call, if and when the class hierarchy changes so that the optimistic assumption is no longer true.

LaTTe also performs a variety of classical compiler optimizations. These include *value numbering* and consequent *common subexpression elimination*, *loop invariant code motion*, *copy* and *constant propagation*, *folding*, and *inlining of static*, *private*, and *final* methods.

LaTTe's unit of optimization is a tree region. These tree regions differ from those used by **DAISY**. **DAISY** produces *output tree regions* based on *PowerPC* code with arbitrary control flow. For example, if *PowerPC* code has a diamond as in Fig. 16(a), **DAISY**'s generates a tree group as output via tail duplication of block **D** as shown in Fig. 16(b). **LaTTe** on the other hand looks for *tree groups in input code*. Thus, as shown in Fig. 16(c), **LaTTe** groups **A–B–C** as one tree group and **D** as another.

LaTTe's register allocation converts the *JVM*'s stack-based model with push and pop operations to the register-

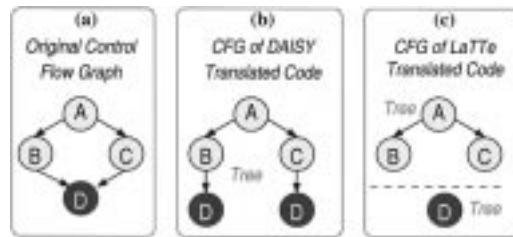


Fig. 16. Tree regions: **DAISY** and **LaTTe**.

based model used in RISC machines such as *Sparc*. The first step in this process is to convert *JVM* push and pop operations to copy operations between elements of an arbitrarily large register set. As many of these copy operations turn out to be unnecessary, **LaTTe** does aggressive copy elimination, which greatly reduces the number of operations compared to what is directly specified by *Java* bytecodes.

Next, **LaTTe** uses its tree regions to provide fast and effective two-pass register allocation. The first pass is a backward sweep, e.g., starting at **B** and **C** in Fig. 16(c) and moving up to **A**. At the tips (**B** and **C**) of the group, **LaTTe** determines the preferred physical register allocations for each live symbolic register, based on the physical registers in which successor groups (such as **D**) expect those symbolic registers to be. For example, if group **D** expects symbolic register x to be in physical register $r7$, and if x is defined in group **A–B–C**, then **LaTTe** will try to allocate the last assignment to x in $r7$. Register allocation preferences are propagated upward through copy operations by the backward sweep phase: e.g., if the last assignment sequence related to x is $z = w + w, \dots, y = z, \dots, x = y$, then **LaTTe** will try to allocated all of z, y, x in $r7$. The backward sweep conveys upward exactly this type of information for all symbolic registers live at group exits. The forward sweep then performs actual register allocation based on the "hints" provided by the backward sweep.

LaTTe's sophisticated JIT compilation techniques take considerably more time than the *Kaffe Java* JIT—12 000 *Sparc* cycles per bytecode versus 4000 *Sparc* cycles per bytecode. However, most *Java* programs such as those in **SPECjvm98** and the **Java Grande** benchmark spend relatively little time in translation and relatively more time in executing translated code—making it important to do a good job at optimization. This is borne out by the fact that **LaTTe** is about $2.2\times$ faster on average than *Kaffe* in running these benchmarks.

We close this section on **LaTTe** as we did Section III on **DAISY**, by noting that **LaTTe** is now available as open source. Details, documentation, and code can be found at: <http://latte.snu.ac.kr>.

VII. DISCUSSION

We have described several leading **BT** projects, as well as some exciting future possibilities for **BT**. We close with a few additional observations.

Sections III and IV provided some details about **DAISY** and **Crusoe**'s *target architectures*. However, they did not

outline what features are *required*. While no architectural feature is required to allow binary translation to be performed, several properties are highly desirable to emulate a particular target architecture *efficiently*. Perhaps most important in this regard is that the *target architecture* have an I/O system and memory map that is compatible with that of the *source architecture*. Without this compatibility, low-level operations writing to the display or disk will not work and the system will not function. For efficiency, it is imperative that the *target architecture* have many other similarities with the *source architecture* including operation semantics, data formats (including condition code and floating point formats), address translation, and special purpose registers such as timers. More registers than the *source architecture* is also desirable so as to be able to speculate and rename results as well as to have scratchpad space in which the **VMM** may work.

As noted in Section I, open problems remain, making **BT** an interesting research area. Among these are the following.

- Can a binary translation machine have generally better performance than a well-designed superscalar?
- Can all real-time problems be avoided?
- What memory management schemes are best over a wide range of TCache sizes?
- In full system **BT**, how can the amount of memory for the **VMM** be transparently (with no operating system knowledge or intervention) increased or decreased after system startup? If the **VMM** gives memory back to the system, it is possible that the OS will note that some previously “failed” part of memory now works well. However if the **VMM** needs more memory—perhaps because of an unusually large working set for the TCache—it has no way to transparently steal the memory from the OS running above it.

Another question is whether the *target architecture* should ever be exposed for users to access directly—bypassing the *source architecture* layer and translation by the **VMM**. Two points in favor of such exposure are that run-time translation overhead is avoided and full (time-consuming) compiler optimizations can be used. However, several factors argue against exposing the *target architecture*.

- If the *target architecture* is exposed, it becomes a *de facto* standard, and becomes difficult to change, just as traditional *source architectures* are difficult to change.
- “Under the covers” optimization and profile directed feedback by the **VMM** is lost.
- The resulting system is complicated, with code for both the *source* and *target architectures*. System administrators and/or the OS must ensure that these disparate pieces work correctly together.
- The target architecture may not support a full set of protection mechanisms in hardware, relying instead on the **VMM** to implement some protection mechanisms. Native target architecture code injected into the platform may introduce security hazards into this **VMM**-mediated security model.

As we have described, dynamic compilation offers many exciting research challenges. These include the possibility of efficiently unifying several existing architectures, and the hope of truly *write-once, run-anywhere* software. We have covered but a small part of the vast field of **BT**, but hope we have piqued the interest of the reader. We encourage you to explore our bibliography and help solve the remaining **BT** challenges.

ACKNOWLEDGMENT

The authors thank the anonymous referee for insightful comments and observations.

REFERENCES

- [1] Intel. (1996) A Tour of the Pentium Pro Processor Microarchitecture. [Online]. Available: <http://pentium.intel.com/procs/p6/p6white/p6white.htm>
- [2] K. Diefendorff, “Power4 focuses on memory bandwidth,” *Microprocessor Rep.*, vol. 13, Oct. 1999.
- [3] R. Hilgendorf and W. Sauer, “Instruction translation for an experimental S/390 processor,” in *Proc. Workshop Binary Translation 2000 (WBT-2000)*, Philadelphia, PA, Oct. 2000.
- [4] A. Klaiber. (2000, Jan.) The technology behind Crusoe processors. Tech. Rep., Transmeta Corp., Santa Clara, CA. [Online]. Available: <http://www.transmeta.com/crusoe/download/pdf/crusotechwp.pdf>
- [5] K. Ebcioglu and E. Altman, “DAISY: Dynamic compilation for 100% architectural compatibility,” IBM T. J. Watson Research Center, Yorktown Heights, NY, Res. Rep. RC20538, 1996.
- [6] V. Bala, E. Duesterwald, and S. Banerjia, “Transparent dynamic optimization: The design and implementation of Dynamo,” HP Laboratories, Cambridge, MA, Tech. Rep. 99-78, June 1999.
- [7] A. Chernoff, M. Herdeg, R. Hookway, C. Reeve, N. Rubin, T. Tye, S. B. Yadavalli, and J. Yates, “FX!32—A profile-directed binary translator,” *IEEE Micro.*, vol. 18, pp. 56–64, Mar. 1998.
- [8] B. S. Yang, S.-M. Moon, S. Park, J. Lee, S. Lee, Y. Chung, S. Kim, K. Ebcioglu, and E. Altman, “LaTTe: A Java VM just-in-time compiler with fast and efficient register allocation,” in *Proc. 1999 Int. Conf. Parallel Architectures and Compilation Techniques*. Newport Beach, CA, Oct. 1999, IFIP WG 10.3, pp. 128–138.
- [9] D. Keppel. (Links to emulation projects). [Online]. Available: <http://www.xsim.com/bib/index2.d/ToDo-9.html>
- [10] J. L. Bower and C. M. Christiansen, “Disruptive technologies: Catching the wave,” *Harvard Bus. Rev.*, pp. 43–53, Jan.–Feb. 1995.
- [11] C. M. Christiansen, *The Innovator’s Dilemma: When New Technologies Cause Great Firms to Fail*. Boston, MA: Harvard Business School Press, 1997.
- [12] M. Gschwind, K. Ebcioglu, E. Altman, and S. Sathaye, “Binary translation and architecture convergence issues for IBM System/390,” in *Proc. Int. Conf. Supercomputing (ICS’00)*. Santa Fe, NM, May 2000, pp. 336–347.
- [13] L. V. Gerstner. (2000, Dec.) Keynote speech given during the E-Business Conference Expo 2000. [Online]. Available: <http://www.ibm.com/lvg/1212.phtml>
- [14] Microsoft. (2000, June) Microsoft.NET: Realizing the next generation Internet. [Online]. Available: <http://www.microsoft.com/business/vision/netwhitepaper.asp>
- [15] Hewlett-Packard. e-Services Defined. [Online]. Available: <http://www.hp.com/solutions1/e-services/understanding/index.html>
- [16] —, e-Utilica Summary. [Online]. Available: <http://www.hp.com/techservers/products/eutilica/summary.html>
- [17] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielsen, S. Thatte, and D. Winer. SOAP: Simple object access protocol. [Online]. Available: <http://msdn.microsoft.com/xml/general/soapspec.asp>
- [18] T. B. Steel, “A first version of UNCOL,” *Proc. JCC*, vol. 19, pp. 371–378, 1961.
- [19] S. Lucco, “Design and efficient implementation of virtual machines,” Tutorial given during the PLDI 1999 Conference <http://www.cs.rutgers.edu/pldi99/tut.html>, May 1999.

- [20] K. Ebcioglu and E. Altman, "DAISY: Dynamic compilation for 100% architectural compatibility," in *Proc. 24th Ann. Int. Symp. Computer Architecture*. Denver, CO, June 1997, pp. 26–37.
- [21] K. Ebcioglu, E. Altman, S. Sathaye, and M. Gschwind, "Execution-based scheduling for VLIW architectures," in *Euro-Par '99 Parallel Processing—5th International Euro-Par Conference*. Berlin, Germany: Springer-Verlag, Aug. 1999, number 1685 in Lecture Notes in Computer Science, pp. 1269–1280.
- [22] —, "Optimizations and oracle parallelism with dynamic translation," in *Proc. of the 32nd ACM/IEEE International Symposium on Microarchitecture*. Haifa, Israel: ACM Press, Nov. 1999.
- [23] K. Ebcioglu, *et al.*, "Some design ideas for a VLIW architecture for sequential-natured software," in *Parallel Processing*, M. Cosnard, *et al.*, Eds. North-Holland, 1988, pp. 3–21.
- [24] J. Moreno and M. Moudgill, "Method and apparatus for reordering of memory operations in a processor," U.S. Patent 5 758 051, May 1998.
- [25] S. Shankland. (2000, Jan.) Transmeta shoots for 700 MHz with new chip. CNET News. [Online]. Available: <http://cnet.com/news/0-1003-200-1 526 340.html?tag=st.ne.ni.rnbot.rn.ni>
- [26] E. Kelly, R. Cmelik, and M. Wing, "Memory controller for a micro-processor for detecting a failure of speculation on the physical nature of a component being addressed," U.S. Patent 5 832 205, Nov. 1998.
- [27] M. Gschwind and E. Altman, "Optimization and precise exceptions in dynamic compilation," in *Proc. Workshop Binary Translation 2000 (WBT-2000)*, Philadelphia, PA, Oct. 2000.
- [28] K. Ebcioglu, M. Kumar, and E. Kronstadt, "Method and apparatus for improving performance of out of sequence load operations in a computer system," U.S. Patent 5 542 075, July 1996.
- [29] K. Ebcioglu, D. Luick, J. Moreno, G. Silberman, and P. Winterfield, "Method and apparatus for reordering memory operations in a superscalar or very long instruction word processor," U.S. Patent 5 625 835, Apr. 1997.
- [30] S.-M. Moon. (2000) The LaTTe virtual machine performance results. [Online]. Available: <http://latte.snu.ac.kr/performance/result.shtml>
- [31] Y. C. Chung, S.-M. Moon, K. Ebcioglu, and D. Sahlin, "Reducing sweep time for a nearly empty heap," in *Proc. ACM SIGPLAN-SIGACT 2000 Symp. Principles of Programming Languages (POPL'00)*, Boston, MA, Jan. 2000, pp. 378–389.
- [32] R. Hilgendorf and W. Sauer, "Instruction translation for an experimental S/390 processor," *Computer Architecture News*, Mar. 2001.
- [33] K. Ebcioglu, "Some design ideas for a VLIW architecture for sequential-natured software," in *Proc. IFIP WG 10.3 Working Conf. Parallel Processing*.
- [34] M. Gschwind and E. Altman, "Optimization and precise exceptions in dynamic compilation," *Comput. Architect. News*, Mar. 2001.



Erik R. Altman received the Ph.D. degree in computer science from McGill University.

He has been a Research Staff Member at the T. J. Watson Research Center, Yorktown Heights, NY, since 1995. Aside from being one of the originators of the **DAISY** project, his research interests include binary translation and optimization, compilers, architecture and microarchitecture. He also contributed to the definition of *Cell*, a next-generation high-performance system architecture, being developed

by IBM, Sony, and Toshiba.



Kemal Ebcioglu received the Ph.D. degree in computer science from the State University of New York at Buffalo in 1986.

He heads the **DAISY** project. He has been conducting research on compilers and architectures for instruction level parallelism topics (in particular VLIW) at the IBM T. J. Watson Research Center, Yorktown Heights, NY, since 1986 (<http://www.research.ibm.com/vliw>). He has many technical publications and patents.

He is the current ACM SIGMICRO chair, the steering committee chair for the Parallel Architectures and Compilation Techniques conference, and the vice president for North America for IFIP Working Group 10.3 (Concurrent Systems). He has served as general chair, program chair, program committee member, and steering committee member for various conferences related to fine grain parallelism. His current research interests include Java, and dynamic binary-to-binary compilation toward achieving high ILP and hardware commonality across architectures.

Dr. Ebcioglu is an Associate Editor of the IEEE TRANSACTIONS ON COMPUTERS.



Michael Gschwind (Senior Member, IEEE) received the M.S. and Ph.D. degrees in computer science from Technische Universität Wien, Vienna, Austria, in 1991 and 1996, respectively.

He is a Research Staff Member at IBM T. J. Watson Research Center, Yorktown Heights, NY. At IBM, he has contributed to several generations of binary translation architectures exploiting instruction-level parallelism, the evaluation of future microarchitecture options for current architectures, and the definition of

the *Cell* next-generation high-performance system architecture. Before joining IBM in 1997, he was a faculty member at the Department of Computer Engineering, Technische Universität Wien, Vienna, Austria. His research interests include computer architecture and microarchitecture, compilers, instruction-level parallelism, hardware/software co-design, application-specific processors, and field-programmable gate arrays. He is the author of over 50 papers and holds several patents on high-performance computer architecture.

Dr. Gschwind is a Senior Member of the IEEE Computer Society, and Member of Phi Kappa Phi and the Fulbright Alumni Association.



Sumedh Sathaye received the Ph.D. degree in computer engineering from North Carolina State University, Raleigh, NC.

He is a Research Staff Member at the IBM T. J. Watson Research Center, Yorktown Heights, NY. His research interests include all aspects of computer architecture and microarchitecture, architectures for broad-band applications, binary translation and dynamic optimization, and compilation for instruction-level parallelism. He has contributed to various binary translation and

optimization research projects in IBM. Most recently he has contributed to and continues to work on the *Cell* high-performance broad-band processor architecture and its applications.