RETROSPECTIVE: DAISY: Dynamic Compilation for 100% Architectural Compatibility

Kemal Ebcioğlu and Erik R. Altman

I. Motivation

DAISY grew out of earlier VLIW experience at IBM Research where promising designs had the Achilles heel of not being able to run existing software. Dynamic and hidden translation of binary code for an existing ISA such as Power^{TM} to a simple VLIW architecture by a just in-time VLIW parallelizer seemed to be an appealing and tractable way to address this compatibility issue.

Both Power and any *migrant* VLIW machine – as the paper termed the target of the exercise – were practically Turing machines of course, so the ability to run Power programs was never in doubt – only the ability to do so with improved performance and low overhead. Low overhead included both software translation overhead and any chip area needed purely for compatibility reasons.

II. DAISY Approach

We began by examining our existing VLIW compiler infrastructure. Prior to DAISY the focus was almost completely on using a static compiler to extract as much ILP as possible from the code to be run. Prior to DAISY that code was in source or intermediate code form as with any other compiler. With DAISY the focus shifted in two ways: (1) Dynamic compilation was used facilitate handling of issues difficult in static code translation, e.g. identifying all possible code fragments or self-modifying code. (2) Instead of maximum ILP extraction, the goal became "good enough". Initial investigations showed promise for "good enough", e.g. operations like global instruction scheduling, loop unrolling, and software pipelining were doable – typically with greedy algorithms – with two orders of magnitude less compiler overhead, while losing only 10% - 20% of potential ILP, as we reported in the paper.

We then moved to address the many challenging architectural areas that a traditional static compiler can ignore. These challenges included self-modifying code, precise exceptions for page faults and other cases, memory-mapped I/O, and strong multi-processor memory consistency. We note that even translating from Power – a relatively modern architecture for the time – self-modifying code was an issue, as it must be for essentially any architecture. For example, Java JITs may produce new and improved code, and even static code pages are "modified" when code is first loaded into the page, or later when completely different code is placed on the same physical page. And because DAISY aimed for "100% architectural compatibility" with Power, it had to handle operating system pages and even pages where address translation is not used – a situation that occurs quite a bit during system boot, another area DAISY had to support for 100% compatibility.

The paper details how we dealt with these challenges, and we do not repeat that here. However, we do note the importance for high-performance of *co-designing* the migrant VLIW, the translation software, and the VMM (Virtual Machine Monitor), all with the original base ISA in mind. In addition , the VLIW targeted by DAISY used the same layout as Power for integer, floating point, and condition code registers, as well as other more obscure but Power-architected registers like XER and FPSCR (integer and floating point exception registers). Although the layout was common, as a feature for increasing parallelism, DAISY used larger numbers of registers than Power, e.g. 64 integer and floating point registers instead of the 32 provided by Power. This increased number allowed us to put speculative results in the 32 registers not provided by Power, and then copy results to the register used by Power in original (binary) program order. This example also illustrates the value of co-design as noted above.

DAISY's dynamic binary translation also revealed things that were much better than compiling from source or intermediate code. Transparent, always-on profiling is an obvious example. A perhaps less obvious case is alias analysis. DAISY did only rudimentary alias analysis, but when in doubt about aliasing DAISY speculated loads above stores – opposite the normal static requirement. As detailed in the paper we could dynamically detect and recover when such speculation failed. In practice it rarely did. In the Spec95 benchmarks available at the time, less than a dozen static load sites accounted for all of the misspeculated loads, and we then just dynamically rescheduled those loads not to speculate.

III. Observations

After the DAISY paper we continued our efforts. We built up the system sufficiently to boot a PowerPC 604e workstation using DAISY and run standard applications on it. That follow-on exercise revealed many hidden corners of the Power architecture, e.g. (1) when is single-byte access required for the string instructions like LSCBX that were in the Power architecture; (2) how to deal with power-on-self-test during boot that turns off all but one memory bank – and hence sometimes powers down the memory used by DAISY; and (3) how to deal with micro-architecture specific registers like

HID0 that can do things such as controlling cache associativty. The presence of such registers mean that at the periphery dynamic binary translation must account not only for the ISA, but the microarchitecture for which it is co-designed. Shortly after the DAISY paper Transmeta had to deal with such issues in emulating the ubiquitous Northbridge and Southbridge used for I/O in the x86 architecture. More broadly, ISA emulation alone is probably not enough for 100% compatibilty: the standard I/O architecture must also be emulated.

In the final analysis, IBM decided not to use DAISY in a product. However, DAISY became a well-publicized research effort of IBM. Furthermore, many of the DAISY techniques were used in a variety of successful offerings from Java JITs to HP's Dynamo offering to speed software on the same architecture. And Transmeta showed the world that dynamic binary translation could be commercially viable, at least for a time.

Looking forward, will dynamic binary translation rise again? With the slowing of scaling and the end of frequency scaling, we see a proliferation of accelerators and even ISAs. Will it be economically important at some point to coalesce this "zoo" of offerings into a smaller number of unified designs? Although the phrase did not make it into our paper, we viewed DAISY as making "ISA a layer of software" and enabling efficient emulation of not just one, but many ISAs on a single migrant design. Perhaps we were just a few decades early in promulgating this view.

A traditional out-of-order processor is in fact an efficient hardware implementation of a run-time interpreter and scheduler of instructions of an ISA. A disruptive research direction taken by DAISY was moving run-time interpretation and scheduling of instructions to compile time, therefore achieving simplification of the target hardware, and allowing new architecture features for increasing parallelism to be freely added to the target hardware without being encumbered by the restrictions of the base ISA. For example, the DAISY VLIW architecture which executes *VLIW tree instructions with conditional execution*, described in figure 1 in the paper, can be seen as an *interpreter* of a simple Mealy finite state machine whose states correspond to VLIW tree instructions. Going further in the direction of replacing interpretation by compilation, a next research stage in binary translation could very well be: at the higher optimization tiers, targeting not a finite state machine interpreter like the DAISY VLIW, but the hardware design of a much wider/larger finite state machine (or other customized hardware) itself, implemented with FPGAs, or, in a future where inexpensive rapid transformation to ASIC becomes possible, with ASICs. The industry has been addressing performance needs of specific applications by manually designing ASICs and optimized software libraries (e.g. TensorFlow), with intense specialized effort, while general-purpose microprocessor designs have been continuing on an evolutionary path. However, we believe that improving the performance of general-purpose code by further removing run-time interpretation and scheduling overheads and adding new features to the target hardware for removing barriers to parallelism, while maintaining compatibility with a base ISA, remains an important research direction.