Daytripper

Dynamic Translation for Intel's Loop Stream Decoder

Nick Black
nickblack@linux.com

Abstract

Intel processors since the 65nm Conroe Core™2 have included hardware to queue decoded loops directly into the out-of-order execution engine. These "Loop Stream Detectors" (LSD) allow for substantial power savings and, in some cases, performance improvements. Unfortunately, the LSD can only store instruction streams meeting a number of architecture-specific restrictions. The Loop Stream Decoder represents an unmistakable power optimization, can be definitively verified via the Core™i7's LSD_UOPS performance counter, and has clearly-defined requirements for successful use. All these properties make optimizing for the LSD an attractive prospect, especially for a runtime translator. Daytripper consists of a DynamoRIO [1] client module capable of discovering loops which fail to engage the LSD, analyzing them for valid transformations, and attempting to reschedule their bodies to take full advantage of this hardware.

Categories and Subject Descriptors D.3.4 [*Processors*]: Compilers

General Terms Binary translation, instruction decoding

Keywords Loop Stream Detector, Length-Changing Prefix, MSROM

1. Introduction

The x86's CISC legacy, unique among modern processors, requires substantial instruction decoding hardware and several pipeline stages. Recent Intel processors provide no less than four instruction decoders, along with an MSROM. Of these, only one ("complex") decoder can handle instructions decoding to multiple μops; for very long instructions (those decoding to six or more μops), this complex decoder must engage the slow Masked-Or ROM unit and its microcode store. This hardware's power requirements, not to mention the havoc wrought on hot loops by front-end stalls, led to the introduction (on Crusoe Core™2 processors) of the original Loop Stream Detector.

Benefits of the original LSD included [6]:

- The documented ability to shut down instruction fetch and branch prediction hardware.

Copyright is held by the author.
CS8803DC May 6, Atlanta.

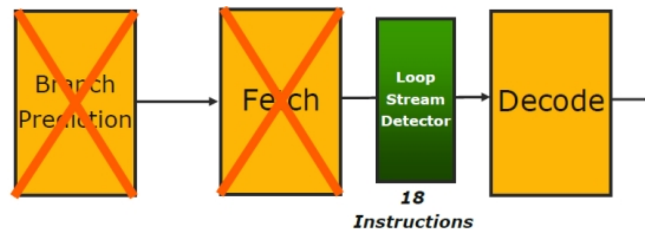

Figure 1. The Core™2 Loop Stream Detector

- The possibility of shutting down instruction cache *in partes* or *in toto*, as explored in other processor designs [2] (no such capability has been mentioned in Intel documentation).
- Elimination of delays due to misaligned branch targets.
- Recovery of execution bandwidth used by branch instructions.

The Loop Stream Detector was improved for the release of the "Nehalem" Core™i7. Moved after the decoding stages, it now supplies μops directly to execution units (as opposed to instructions to the decoder).

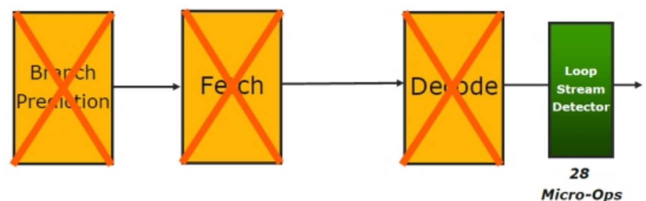

Figure 2. The Core™i7 Loop Stream Detector

Benefits include:

- The entire processor frontend can be powered down during LSD streaming, as opposed to merely the instruction fetching hardware.
- Length-Changing Prefix stalls, major sources of delays in the frontend, are eliminated.
- Stalls due to contention for the single complex instruction decoder are avoided, as are the extreme delays due to MSROM-based decoding.

The Loop Stream Decoder is a microarchitectural property: it operates wholly without programmer intervention, and is not visible in the x86 ISA. The LSD will cache any instruction/μop stream that has looped (branched backwards) 64 times, and meets the following conditions:

- It requires no more than 4 instruction fetches of 16 aligned bytes each.

- It contains no more than 4 branches, none of them a CALL or RET.
- It contains no more than 18 instructions (on Core™2).
- It contains no more than 28 μops (on Core™i7).

2. Optimizing for the LSD

Optimizing for the LSD requires the abilities to recognize hot loops, determine whether or not a given loop is suitable for the LSD, and transform unsuitable loops into suitable forms. This last must be performed while honoring the original program's semantics, and (obviously) in a fashion guaranteed to terminate. Daytripper leverages the DynamoRIO binary translation framework for disassembly and encoding of instruction streams, and its *trace* objects (as opposed to *blocks*) provide suitable recognition of hot loops.

2.1 Determining LSD suitability

Determining whether or not a loop qualifies for the LSD can be a difficult task in and of itself. The requirements of the Core™2's LSD are simple enough for a static compiler to consider, but the Core™i7's LSD complicates matters by caching μops rather than instructions. Intel does not (at this time) make public the correspondence of instructions to μops[1]. These mappings change from processor to processor or even stepping to stepping, and are further mudded by microarchitectural techniques such as Micro- and Macro-fusion. Any compiler must already have some idea of μop mappings if it is to fully optimize for decoder resources, but simply knowing which instructions decode to multiple μops is sufficient to make full use of Pentium-M, NetBurst, Core and Nehalem microarchitectures' "3-1-1" or "4-1-1-1" decoder arrangements.

Thankfully, a performance monitoring counter (LSD_UOPS, event 0xA8) was introduced alongside the improved LSD. While a static compiler (especially in the absence of profile-guided optimization) would be hard-pressed to effectively make use of this PMC as a litmus (especially as runtime code placement decisions can affect suitability), a dynamic compiler is well-suited to begin its search with the counter. Daytripper simply resets the counter on entrance to a DynamoRIO trace entry callback, and checks it during the trace exit. If the counter has changed, the LSD has been employed, and Daytripper needn't perform further work. Likewise, a zero count suggests analysis of the associated trace.

The performance counter also serves to gauge our effectiveness. Daytripper marks any trace it modifies. Since it never transforms a trace which triggered the LSD, any marked trace with a non-zero LSD count represents a successful transformation[2]. Other marked traces represent wasted work, and are indicative of bugs in Daytripper's suitability testing. Given this instant and highly accurate feedback, any errors in μop mapping ought be flushed out quickly. Daytripper is thus fairly resilient against microarchitectural changes[3], or at least automatically discovers any which affect its effectiveness. Furthermore, this allows binaries to be sorted based on whether they can effectively be transformed (subject, of course, to the same vagaries which complicate static PGO). Daytripper translation can thus be foregone when it will have no effect.

Verifying other properties is a fairly simple (if distinctly unpleasant) task. This merely requires knowledge of branch instructions (at the ISA level) and their LSD limits, location of the code at runtime, disallowed instructions, and properties of the instruction fetch unit. These last are microarchitectural properties, but thankfully not a maintenance burden: the CPUID instruction allows all relevant properties (page size, instruction cache size, and associativity)[4] to be discovered at runtime.

[1] Agner Fog's venerable datasheets [3] do publish an approximate correspondence, but these follow processor releases, at best, by several months.

[2] This doesn't hold if DynamoRIO itself triggers the LSD.

[3] The author spent significant time planning integrating such a scheme into GCC's PGO, thinking it the coolest part of this project.

2.2 Extracting LSD suitability

Transformation, as could be expected, is the most complicated aspect of Daytripper's operation. More correctly, transformation *will be* the most complicated aspect; Daytripper, originally a static binary translator, was reinvented as a dynamic tool very recently. Such transformations are, for all intents and purposes, beyond the capability of a static x86 translator. The ubiquity of indirect branches (even discounting clear linker patch-up points, as can be determined via cross-referencing ELF's `.plt` section) would be sufficient to exclude most transformations[9]; the x86's variable instruction length, relaxed alignment requirements for execution and access, and support for self-modifying code quickly render the problem intractable, if it is indeed even decidable.

3. Loop size reduction

For each requirement of the LSD, there exists a corresponding class of transformations we might perform. Many of them would already have been executed by a reasonable static compiler, and thus we oughtn't expect them to be exploitable in optimized code. This will be of critical importance when selecting benchmarking tools; it'll likely be best to build up a unit testing infrastructure around small, hand-written assembly snippets.

3.1 Code requires more than 4 fetches.

First, ensure that the loop is properly aligned (this will generally be true for optimized code). Otherwise, attempt loop size reduction as outlined in section 3.3. Note that instructions must be minimized in 3.3, even if the LSD is μop-based; it might thus be necessary to minimize *both* μops and instructions on Core™i7 processors and later.

3.2 Code contains more than 4 branches.

Attempt to replace branches with predicated instructions. Daytripper is only active on processors with Loop Stream Detectors, all of which include the P6-era `CMOVxx`; it is thus safe to conditionalize where possible (it is unfortunate that x86 is not a more fully predicated instruction set, such as ARM [11] or IA64). This transformation is unlikely to slow down the loop body, and thus again we can expect optimized code to already have used `CMOVxx` where applicable. Compilers optimizing for size or speed, however, are unlikely to use certain bit-parallelism tricks [12] we might exploit, especially in highly idiomatic arithmetic loops. This might be fertile ground for future research.

3.3 Code is too large

For Core™2, or to reduce the number of necessary fetches (see 3.1), we seek to minimize instructions. For Core™i7 and beyond, we seek to minimize μops. These goals are not incompatible, but neither can they generally be achieved via the same transformations. Unoptimized code can of course be easily shrunk, but we assume reasonable optimization for space—in which case we are unlikely to achieve much reduction—or speed, which has as its most basic heuristic "minimize dynamic instructions". Once again, however, we can perform some "regressive transformations" which possibly sacrifice performance.

[4] Whether the instruction and data caches are unified is irrelevant; we're verifying not residency, but boundary crossings.

Whether to do so must be decided considering the benefits of the Loop Stream Detector. Loops containing length-changing prefixes, for instance, incur weighty stalls in the decoding logic. By bypassing the decoders (on Core™i7), the LSD may well effect a net speedup even as it undoes other optimizations. These stalls have long been known to the compiler-writing community, so it is unlikely that they're generated in any real abundance. If code were found that had been forced to incur LCP stalls so that some other powerful optimization could be performed, Daytripper could unleash potent performance gains indeed.

"Magic divisions" trading size for speed [8] could be replaced with their equivalent constant divisions. `ADD/SUB` $\# - 1$ instructions could be replaced with their equivalent `INC/DEC`s, especially if the condition register-carried dependencies these instructions introduce were demonstrated to be innocuous (perhaps by introducing a dependency-breaking instruction). Sets of `PUSH`es and `POP`s, used to minimize the number of stack writes, could be replaced with a single `PUSHA/POPA` pair. This is unlikely to affect performance negatively, due to a large ratio of cacheline to word lengths (it ought be noted, however, that these are μop-intensive instructions). In any case, extensive pushing and popping is unlikely to occur in a hot, reasonably-optimized loop.

If the 4-fetch limit is not close to being breached (in an extreme case, 18 single-byte instructions), a counter-intuitive method would be to eliminate internal NOPs present only for alignment purposes. Since the LSD eliminates misaligned instruction penalties, these NOPs have no purpose in a loop streamed from the LSD.

4. Related work

Virtually every reference to the Loop Stream Detector, from GCC bug reports [4] to various optimization guides, speaks of supposed performance benefits.

This is wrongheaded.

While it is true that streaming instructions (in the case of Conroe) or μops (in the case of Nehalem) from the Loop Stream Detector bypasses several pipeline stages, this does not, by itself, represent a gain in throughput. At a saturated steady state, IPC is independent of pipeline length. The LSD applies only to tight loops—precisely the sections most easily benefited by branch prediction, large data caches, advanced prefetching and extensive speculation. In short, it targets code for which Intel has already spent a decade adding hardware support (as noted earlier, the LSD *can* remedy certain decoding perversions specific to the x86 architecture).

The LSD would be highly relevant to research on optimizing for power, were it not for the facts that:

- It is present only on energy-hungry high-end x86 processors, poor fits for systems designed to conserve power.
- It is only so effective at reducing power consumption due to the large transistor budget required for high-speed x86 instruction decoding; a classic RISC processor could not reap nearly such significant benefits.

Whether or not the Loop Stream Detector will emerge as the focus of academic research is debatable. This paper appears to be the first investigation of dynamic translation for the explicit purpose of engaging the LSD.

A. Using the LSD_UOPS PMC

Existing open source hardware profiling tools have yet to fully support the Core™i7 line of processors. The line represents several combinations of Family and Model numbers [5]. The following information is valid as of 2010–05–06.

A.1 Linux's `perf`

The `perf` tool, included in Linux source distributions since 2.6.31, can support LSD_UOPS via use of a "raw counter". Provide

```
-e -r 1a8
```

when an event needs be specified; this selects unit mask 0x1 from PMC identifier 0xa8, designating the LSD_UOPS event [7].

A.2 Oprofile

Oprofile requires a patch to properly identify some Core™i7 processors. The author has submitted it to the Oprofile team; it can be found at:

`http://marc.info/?l=linux-kernel&m=127294830417492`.

B. Using DynamoRIO with Daytripper

DynamoRIO requires a patch to properly identify some Core™i7 processors. The author has submitted it to the DynamoRIO team; it can be found at:

`http://groups.google.com/group/dynamorio-users/browse_thread/thread/72dd27ca8f5ead66`

Acknowledgments

I am indebted to David Kanter for his excellent comparison of the Loop Stream Detector to the Pentium 4's trace cache [10].

I stole my images from Ars Technica.

References

[1] DynamoRIO. `http://dynamorio.org/history.html`, 2002–.

[2] A.-M. Badulescu and A. Veidenbaum. Power efficient instruction cache for wide-issue processors. In *Innovative Architecture for Future Generation High-Performance Processors and Systems*, January 2001.

[3] A. Fog. *Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs*. Copenhagen University College of Engineering, 1996–2010.

[4] GCC Bugzilla. Bug 38306 [4.4/4.5/4.6 regression] 15% slowdown w.r.t. 4.3 of computational kernel on some architectures. `http://gcc.gnu.org/bugzilla/show_bug.cgi?id=38306`.

[5] Intel Corporation. *Intel® Processor Identification and the CPUID Instruction*. Application Note 485. August 2009.

[6] Intel Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. Number 248966-018. March 2009.

[7] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Number 253669-033US. December 2009.

[8] D. E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley Professional, November 1997.

[9] W. Landi. Undecidability of static analysis. In *ACM Letters on Programming Languages and Systems*, December 1992.

[10] Real World Technologies. David Kanter. Inside Nehalem: Intel's future processor and system. `http://realworldtech.com/page.cfm?ArticleID=RWT040208182719`.

[11] D. Seal. *ARM Architecture Reference Manual*. Addison-Wesley Professional, second edition, January 2002.

[12] H. S. Warren. *Hacker's Delight*. Addison-Wesley Professional, July 2002.