The Mill: Split-stream encoding

Out-of-the-Box Computing
ootbcomp.com

Abstract –
Real-world programs often thrash in the instruction cache, especially when SMT methods are used. The Mill™ split-stream encoding doubles the effective capacity of the instruction cache at no increase in per-instruction power usage or cache access latency, while also sharply increasing the potential maximal decode rate for instruction sets that use variable-length encoding.

Keywords: split-stream encoding, instruction cache, instruction decode

Introduction

CPU architectures never have enough instruction cache. Benchmarks and demos may fit, but real-world workloads often thrash: each line loaded to cache evicts another soon-needed line, and program progress drops to L2 speeds. Yet cache sizes cannot be increased; larger caches require stronger drivers using more power, and the greater distance to the decoder increases latency or lowers clock rate.

Variable-length instruction encoding, as used in the x86 architecture, can increase the effective cache capacity by fitting more instructions into a fixed cache byte size. However, variable-length instructions are hard to parse; power, circuit area, and time constraints conspire to limit parallel variable-length parsing to 4 or 5 instructions. Fixed-length encodings, as used on the Itanium and RISC architectures, have no parallelism constraints but are spendthrift of bits and are quicker to thrash in cache.

Terminology used for instruction encoding in the literature is inconsistent. As used here:

- An ***operation*** is the unit of execution, such as an individual add, load, or branch.
- A ***bundle*** is the unit of physical encoding; decoders receive bundles to decode.
- An ***instruction*** is the unit of logical encoding; all actions of an instruction are semantically performed together.

In many architectures a single chunk of bits carries all three of these meanings: the bits are both the logical and physical unit for decode and carry only a single operation command. VLIW and other wide-issue encodings pack several operations into a single bundle that constitutes the entire instruction. In the Itanium, bundles hold exactly three operations, while an instruction may contain a variable number of operations, and the encoding of a single instruction may cross bundle boundaries.

A Mill instruction comprises exactly two variable-length bundles, each of which may encode a large number of variable-length operations. In addition, the encoding permits the instructions to contain meta-data used to aid decoding or execution of the operations.

Combining these and other innovations, the Mill is able to decode over 30 operations per cycle, five to ten times as many as contemporary legacy architectures. This decode throughput is needed for high-end members of the Mill family (which also execute the operations at the same rate) when software-pipelining a loop. Low-end Mill members (with much less compute capacity) do not need as much decode throughput but still benefit from the increased cache capacity provided by the same encoding methods.

Split-stream encoding

Imagine, as in Figure 1, an encoding in which each instruction comprises a single bundle

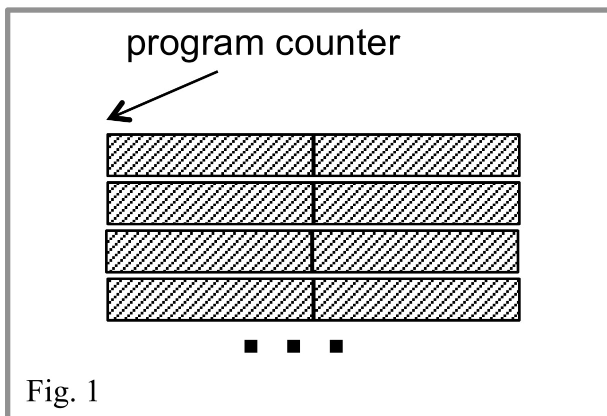

Fig. 1

containing two operations, and a program comprises many sequences of these bundles connected by flow-of-control operations. A two-wide VLIW might use such an organization.

A sequence is entered at the head by a branch from some other sequence, and will exit via another branch after executing some portion of the sequence. Sequences like this, with a single entry point and possibly several exits, are commonly called ***superblocks***, ***extended basic blocks*** or ***EBBs***.

A branch operation contains the address of its EBB target, so when a transfer occurs the decoder knows where the target EBB begins and can start decode of the first operation of the first bundle. However, in variable-length encodings the decoder does not know where the second operation in that bundle begins, nor where the following bundle begins. Hence the decoder must first discover the length of the first operation before it can begin decoding the following operation, and must discover the lengths of both operations before it can begin decode of the following bundle.

Instruction decoders routinely discover the length of an operation in a single cycle. To discover two lengths, one after the other, is more difficult, and more than two is impractical at modern clock rates. Rather than parsing one operation at a time, some CPUs contain several different decoders and start one at each of several possible operation boundaries before knowing where the boundary actually is. Necessarily, several of these decoders will be working on misaligned bit sequences that aren't really an operation. Later, when lengths are known, the results of the misguided decoders are discarded and only the results that started on the correct boundary are kept and executed. This approach is costly in power and chip area, but permits parallel decode of sequences of up to five or so operations.

It is necessary to reorganize the encoding to be able to decode more operations per cycle. For example, imagine that the program has been reorganized as in Figure 2. What had been a single sequence of two-operation bundles is now two sequences of one-operation bundles. The first new sequence comprises all the first operations of the former bundles, and the second sequence comprises all the second operations. The same operations are present as before, merely reorganized in memory.

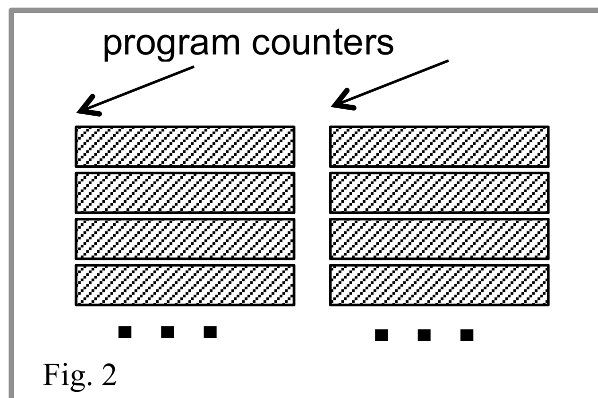

Fig. 2

A decoder for such an organization takes in one bundle from each sequence every cycle, and may be more clearly thought of as two decoders, one for each sequence, that decode the two streams in lock step. Each stream decoder need only determine the length of a single operation to locate the start of the following bundle, and has a whole cycle to do so.

Similarly, splitting the EBB across two streams may be more clearly thought of as splitting it into two EBBs that are decoded in lock-step. Obviously this approach will work for any degree of splitting, not just into two. A stream of bundles of seven operations could be split into seven streams of one-operation bundles just as easily. Note that each of the split streams has its own program counter that contains the address of the

current bundle of that stream and will be incremented to the address of the following bundle.

Once the split decoders are started on a split EBB, they can decode at full speed until the program executes a taken branch. At that point, the decoders must each be given the address of the target of their split stream; for a two-way split they need two target addresses, seven for a seven-way split, and so on.

This presents a problem. While encoding two target addresses in a branch operation is marginally possible, the space occupied by the second address would penalize the encoding efficiency, and seven such addresses are quite out of the question.

However, if the EBB is split exactly two ways, then the two heads can share a single address, as in Figure 3. The two EBB streams are butted

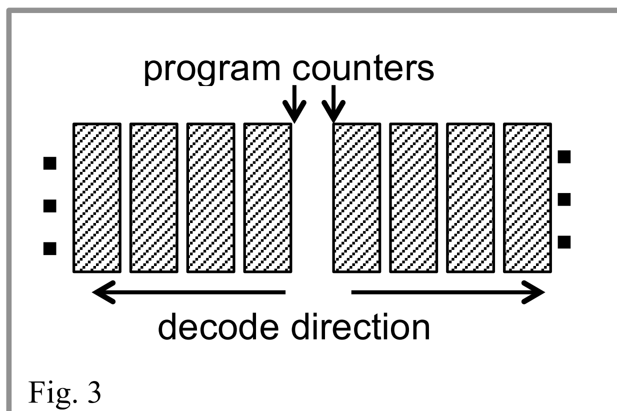

Fig. 3

together head-to-head, with one stream extending toward higher addresses and one extending toward lower addresses. In this organization, a branch operation contains a single target address which applies to both streams. A transfer is to an ***entry address*** located in between the two half-EBBs. Once transferred, each decoder advances by one bundle each cycle, both moving away from each other in opposite directions. This is ***split-stream encoding***.

This bi-directional decode is of course limited to two streams. Alternatively, it is possible to support more than two streams with a single entry point if the streams are all decoded in the same direction and are interleaved cache line by cache line in memory; stream one would use the first line after the entry, stream two the second, and so on, repeating round-robin fashion. Each decoder could find its sequence of lines from the starting entry point, and each sequence would be kept in its own cache. However, it is unlikely that all the streams would comprise the same number of lines, so variations of stream length within and EBB would lead to internal fragmentation of the memory used.

While branch and function call operations target a new EBB and need only a single target address for both split streams, a return operation in split-stream encoding must resume decode at the point of call in two different streams. The call operation saves two return addresses in the stack or other structure, and the return operation restores the decoders to those addresses.

If the operations are weighted by their difficulty to decode and are evenly balanced across the two streams then split-stream cuts the load on either individual decoder in half. Actually, the difficulty is cut by more than that when the instruction set uses variable-length encoding, because cost of decode is much worse than linear in the number of operations to decode.

Moreover, if operations with similar encoding requirements are assigned to the same stream, then the bundle or operation formats can be tailored to each stream, saving instruction space and likely reducing decoding difficulty. For example, all operations with an effective-address parameter (including loads, stores, and branches) can be assigned to one of the streams, so the logic to decode an address is not needed in the other decoder.

In conventional legacy architectures with a single instruction stream, the instructions reside in memory and are cached in a small, fast access instruction cache that is all-too-frequently too small for the working set of the program. In split-stream encoding, the cache can be replicated so that each stream has its own instruction cache.

If each of the split caches is the size of a single conventional cache, then the pair provides double the cache capacity for instructions. In addition, by specializing the encoding format for each stream, the average length of instructions can be reduced,

thereby increasing the effective cache capacity by an amount that depends on the format details.

Having two caches does not increase latency because each cache can be located near the corresponding decoder just as a single cache could be placed near the single decoder in a conventional single-stream architecture. For the same reason, power consumption per operation is also unchanged by the split. The only cost for the doubled cache capacity is the increase in chip area for the second cache, which may cause a small drop in chip yield.

Instruction caches are organized as a collection of fixed-length cache lines, typically capable of containing several instruction bundles. Because a split EBB shares a common entry address across both streams, the line containing the entry point will be needed by both decoders when the EBB is entered. A naive cache replacement policy would cause the entry line to reside in both caches, wasting space.

Instead, the entry line resides in only one of the caches, as selected by the low-order significant bit of the entry address. This effectively randomizes the residence of the entry lines. When a branch occurs, the target address determines which cache is queried for the entry line, which is fed to both decoders. Depending on the physical layout of the caches and decoders, the entry line may take an extra cycle to reach the non-local decoder. After the entry line, subsequent lines are unshared and reside only in the cache for their stream. Thus the two caches can feed the decoders with minimum latency.

The two caches should be of the same size if the operations assigned to each stream produce roughly similar demand for bytes. If the encoding and/or operation assignment leads to imbalanced demand then the two caches may have appropriately different sizes. The larger may have the same number of lines as the smaller but with a larger line size, or may have more lines of the same size. Which strategy will perform better in practice depends on the details of the demand and the structure of the memory hierarchy.

Each transfer to a new EBB will on the average discard half a cache line (the bytes after the branching bundle), which was fetched from memory but is not part of the executed instruction stream. In split-stream encoding there are two such wasted half-lines. However, transfers in single-stream encoding also waste half a line at the target (the bytes before the entry point). This is not wasted in split-stream encoding. Consequently, both single- and split-stream encodings waste two half-line fetches at each transfer, although they are different half-lines.

Some legacy organizations place more than just instructions in instruction memory (examples include jump tables and alignment padding). This metadata cannot be meaningfully decoded and so cannot be in the instruction stream proper; instead, it is placed after unconditional branches or ahead of branch targets so that it is isolated from regular control flow. When split-stream is used, then such metadata can be placed at the entry point between the two EBB heads so long as it has a fixed known length or its size can be determined quickly enough not to delay bundle parsing.

Concrete example – the Mill

The Mill™ is a new general purpose CPU architecture family oriented toward high single-thread performance at minimal power consumption. It is a wide-issue exposed-pipeline machine, with sustained decode, issue, and execute rates ranging from five to over thirty MIMD operations per cycle. These rates would be impossible using conventional instruction encoding and decoding strategies.

The Mill uses split-stream encoding. One stream is dedicated to operations that need large manifest constants: loads and stores, flow of control, and large (up to 128 bit) program constants. The other stream comprises all ordinary arithmetic operations, plus some operations unique to the Mill.

Because operations with an address occupy many more bits than simple arithmetic operations, the byte demand on the caches of the two streams is roughly equal despite there being many more arithmetic operations than addressing operations. Consequently the Mill uses equal sized instruction

caches for the two streams.

The Mill uses one byte of metadata at the EBB entry line between the two stream heads. The byte contains counts of the cache lines comprising each of the two EBBs. These counts guide prefetch of instruction cache lines, and prevent the instruction fetcher from running off the end of the EBB in the absence of correct branch prediction.

Prior work

The decoupled access-execute architecture[1] (DAE) used two logical instruction streams to feed two execution engines which ran asynchronously to each other and communicated via hardware queues. Each stream could be independently branched in the conceptual model, with the ability of each processor to wait on a branch in the other. However, in the physical implementation in the Astranautics ZS-1, the streams were physically interleaved with a single branch for both. DAE differs from the methods described here in that DAE has conceptually asynchronous streams intended to hide memory latency, whereas split-stream encoding has a conceptually single stream that is split for encoding efficiency and decode throughput.

- 1 Smith, J.E. "Decoupled access/execute computer architectures", Computer Systems, ACM Transactions on; Volume 2, Issue 4, November 1984, Pages 289-308.