A CLARIFICATION OF THE DYNAMIC/STATIC INTERFACE

Stephen W. Melvin
Yale N. Patt

Computer Science Division
University of California, Berkeley
Berkeley, CA 94720

ABSTRACT

Much has been written about the interface between hardware and software and the tradeoffs associated with it. In this paper, we show that there exists a separate interface which is commonly confused with the hardware/software interface (HSI). This is the dynamic/static interface (DSI), which defines the boundary between interpretation and translation. In this paper we show that the DSI is separate from the HSI and demonstrate the usefulness of this concept.

1. INTRODUCTION

A computer is a multilevel system with problems at the top and circuits at the bottom. In between are levels, or interfaces, which define sets of data structures and the operations allowed on them. For example, high level languages, machine architectures and microarchitectures are all interfaces.

A widely discussed interface is the hardware/software interface, or HSI, which defines the boundary between "hardware" and "software". Another interface, not as widely discussed but equally if not more important is the dynamic/static interface, or DSI, which defines the boundary between translation and interpretation. We will show that the DSI and the HSI are often confused, but are actually separate interfaces. Furthermore, we will show that by clarifying the concepts of the DSI and the HSI, different approaches to computer architecture can be put into a new perspective that we believe is useful. We will also discuss two other interfaces, the single cycle interface, or SCI, and the builder/user interface, or BUI.

This paper is divided into four sections. In this section we define the DSI, HSI, BUI and SCI and discuss general tradeoffs associated with the DSI. Section 2 provides some historical background on the DSI concept. Section 3 discusses various examples in terms of their interface configurations. Section 4 concludes with some final remarks.

1.1. DEFINITIONS

The dynamic/static interface arises from the fact that problem solutions typically undergo two stages. In the first stage, translation, the specification of the problem is changed from one form into another (i.e., a new problem specification is created at a lower interface). In the second stage, interpretation, the problem specification is executed, possibly using input data that is not part of the specification, and results are generated. The DSI is the interface between translation and interpretation.

For conventional machines running compiled languages, the DSI is at the machine architecture level. The high level language is translated into machine language, which then gets interpreted by the hardware. Because this case is so common, the DSI tends to be implicitly placed at the machine architecture level. Often a discussion of "hardware vs. software" will include points that are really related to the issue of static vs. dynamic.

Now suppose we have a program that interprets a high level language. In this case, the DSI is above the machine architecture level. The interpreter is clearly software, so the DSI must be different from the hardware/software interface. Conversely, hardware can be built which translates a program from one interface to another. In this case, the DSI is below the hardware/software interface.

The specific definition of the HSI presents some difficulties. The problem is that the question of what is "hardware" and what is "software" doesn't have a simple answer. The extremes are easy to identify, but the distinction is less clear in between.

We believe that the crucial element in the way most people interpret software and hardware is the question of alterability. In other words, how easy it is to alter a particular interface is related to how "soft" the interface
is. In this paper, we will use the following definition for the HSI: the highest interface that is not dynamically alterable. That is, if an interface cannot be changed while the computer is operating, everything below will be called hardware. Note that we could also have used a test of static alterability (i.e., can the interface be changed without physically modifying the computer?). Perhaps the HSI should be broken down into the hardware/firmware interface and the firmware/software interface and the tests of static and dynamic alterability should be applied to each interface respectively.

Under the dynamic alterability definition of the HSI, the microcode of most machines, even though it may be stored in read/write memory, is hardware because it cannot be changed without halting the processor. Even the microcode of the IBM System/370 model 145, which is stored in main memory, would be considered hardware under this definition because the memory region containing the microcode is protected and cannot be changed without rebooting [17]. However, the microcode of machines such as the B1700 [21] and the QM-1 [28] is software under this definition because it can be modified while the processor is running.

There are also ways to define hardware and software that have nothing to do with alterability. Patt and Ahlstrom [22] argue that microcode should be considered hardware if it is provided by the manufacturer and software if it is written by the user. They have called the hardware/software interface what might be more appropriately called the builder/user interface, or BUI. This interface defines the boundary between what the builder provides and what the user has access to. This definition of hardware and software, however, was rejected by a Federal District Judge in the recent Intel/NEC case. The microcode of the 8086 was ruled to be software even though it is not visible to the user.

Another interface that is useful to discuss is the single cycle interface, or SCI. This is the lowest interface of a machine; it is generally interpreted by combinational logic only. In the case of a pipelined implementation, there may be sequential logic below the SCI, but a particular piece of the data path is not used more than once for each instruction at the SCI level. For conventional machines, the SCI is the microarchitecture (i.e., the microword format, the definitions of the microorders and the definitions of the internal registers).

1.2. DSI TRADEOFFS

There are many tradeoffs associated with the DSI. We are mainly interested in how movement of the DSI affects performance, but it is necessary to understand that performance is variable even with a fixed DSI (i.e., translation time can be traded off for interpretation time). The translator can be very complex, performing many optimizations and making the interpretation phase faster, or it can be very simple, making the interpretation phase slower. In the extreme case, if a problem needs no input data, the translator could do what amounts to executing the entire program and then generate just the statements that print the result.

Movement of the DSI has different implications. If the DSI is at a suitably matched high semantic level, then the translation process would be simple but interpretation would be complex and might require several levels. In this case, the highest level interpreter would be executing instructions which would themselves be interpreted by a lower level interpreter.

On the other hand, if the DSI is at a very low semantic level, for example at the SCI, then there is no interpretation involved except by the circuits below the SCI. Translating to this interface involves several issues. Depending on the complexity of the SCI, translating near optimally may be very difficult. Translating to a simple SCI is easier, but performance may be related to the complexity of the SCI (e.g., how many things can be done in one cycle). Note that the problems associated with translating to the SCI are different from the problems associated with writing an interpreter which executes above the SCI. Thus, it is not always straightforward to compare the compilation to microcode with the creation of a macrocode interpreter in microcode.

There are also bandwidth tradeoffs. Lowering the DSI increases the run-time bandwidth to the highest level interpreter. Raising the DSI would lower the bandwidth required to this interpreter, and since the lower level interpreters are generally smaller, this could be advantageous.

2. HISTORICAL BACKGROUND

The concept of hardware and software and the concept of interpretation and translation have both been around for many years. However, the connection of the two in a coherent manner has not. In this section, we will provide a brief background on how others have viewed the DSI and the HSI and to what extent they have connected the two.

People have long recognized the two-phase nature of the execution of most programs. Hoewel [16] addresses the DSI directly and argues that it should be above the SCI but below the high level language. Flynn [8],[9] also distinguishes the DSI. These papers, however, don't discuss the DSI in connection with the HSI.

Myers, in chapter 3 of [20], compares some basic approaches to computer architecture. He distinguishes five approaches: traditional, language-directed, type A HLL machines, type B HLL machines and type C HLL machines. The main feature separating these ap-
proaches is the level of the DSI, not the level of the
HSI. The discussion centers on the translation and in-
terpretation process, even though the terms "machine
architecture" and "machine language" are used. The
implicit assumption is made that the HSI and DSI are
the same with the exception of type B HLL machines,
which differ from type A machines only in that the HSI
is higher (above the DSI).

Thus, he discusses a category in which the hardware
translates as well as interprets, but he seems to conside-
"Note that the type B machine has the same
semantic gap as a type A machine. Its only
advantage over a type A machine is that the
assembly process should be faster because it
is implemented as a microprogram or in hard-
ware." [20], p. 46

This would imply that he considers the DSI to define
the semantic gap. However, in his discussion of hard-
ware vs. software, he is clearly talking about something
else:

"Architects often use the following three crite-
rnia in determining whether a function should
be implemented in the machine rather than
in software: (1) the function should be small,
(2) the function should be unlikely to change,
and (3) system performance would suffer from
a slower software implementation of the func-
tion." [20], p.46

These criteria are not related to the translation and
interpretation issue and the second criterion clearly re-
lates to a question of alterability. Thus, Myers shows
that the HSI and the DSI (by our definitions) are sepa-
rate things, even though he doesn't discuss them in this
way.

Tanenbaum [34] separates the DSI from the HSI more
clearly although he doesn't connect the two together.
In Chapter 1, multilevel machines are introduced and
the techniques of translation and interpretation are de-
defined. He doesn't mention the DSI explicitly, but he
does discuss translating to and interpreting from differ-
ent levels (i.e., movement of the DSI). Then, software
and hardware are discussed:

"Any operation performed by software can
also be built directly into the hardware and
any instruction executed by the hardware can
also be simulated in software. The decision
to put certain functions in the hardware and
others in the software is made on the basis of
such factors as cost, speed, reliability, and fre-
quency of expected changes." [34], p. 11

Thus, although the DSI and the HSI are not related to
one another, they are clearly considered to be separate
interfaces.

3. DSI/HSI CONFIGURATION EXAMPLES

Figure 1 shows an HSI versus DSI diagram showing
where a number of computers fall within the two dimen-
sional space. In this section we consider different ap-
proaches to computer design and discuss them in terms of
how they affect the HSI and the DSI.

3.1. HIGH LEVEL LANGUAGE MACHINES

The principle behind the notion of high level language
machines is to raise the HSI to the high level language
level. The DSI is also generally raised as high as possi-
ble, usually slightly below the HSI. The SYMBOL ma-
chine was an early example of this [32], [33], [31]. In
this case, the HSI is at the SYMBOL language level
[7] since the hardware is able to accept SYMBOL lan-
guage input, while the DSI is at a slightly lower level
represented by the internal representation of a SYM-
BOL program. The hardware of the machine trans-
lates a SYMBOL program into this intermediate form
/removes redundant blanks, changes keywords into bit
strings, replaces symbolic addresses to pointers, etc.).
After this translation has been completed for the entire
program, execution begins.

Another example of a high level language machine is the
Abacus machine that ran BASIC [3]. Like SYMBOL,
this machine has the HSI at the high level language
level and the DSI slightly below. Abacus did a hard-
ware translation similar to that performed by SYMBOL
before starting the execution of the program. A FOR-
TRAN machine [1] also falls into this general category.
The HSI is at the FORTRAN level, the DSI is slightly
below. Finally, there is a machine proposed by Chu and
Abrahms [5], [4]. Unlike the previous examples, this
machine actually has the DSI at the high level language
level. During the execution of the program, the hard-
ware actually scans the source code and executes it. No
translation is done previous to this and there exists no
other specification of the program other than the source
code.

3.2. DYNAMIC MICROPROGRAMMING

The basic idea of dynamic microprogramming is to low-
ner the HSI to the single cycle level, while keeping
the DSI at a typical level. The motivation behind dy-
namic microprogramming is to allow the DSI to be bet-
ter matched to the problem being run. By lowering the
HSI, the level in between the HSI and the DSI becomes
dynamically alterable, thus, the DSI can be changed
for different problems being run. Examples of dynamic microprogramming are the Burroughs B1700 series [21] and the QM-1 [28].

Cook and Flynn describe a dynamically microprogrammable computer in [6], and Flynn, Neuhauser and McClure describe the EMMY system at Stanford in [10], which was similar. Also, the 'Interpreter' is described in [30], a system similar to the QM-1. Rauscher and Agrawala discuss the application of dynamically microprogrammable machines in [29]. Most of these papers and books, however, don't explicitly address the issue of DSI placement. The main point behind dynamic microprogramming is HSI placement, not DSI placement. It is generally assumed that the DSI should be where it typically has been. In other words, these machines are generally designed as interpreters, they are not set up to handle placement of the DSI at the HSI (compiling down to the microcode level).

### 3.3. VERTICAL MIGRATION

Another idea concerning the movement of the HSI and DSI is vertical migration. The principle here is to raise the HSI. In some cases the DSI is raised with the HSI, and in some cases it remains unchanged. The difference lies in whether the functions that are vertically migrated are then translated to, or are then used by an interpreter (see figure 1). In either case, the motivation behind this idea is to raise the HSI in order to allow more efficient execution of commonly executed functions. This is accomplished for two reasons. First, instruction bandwidth is reduced because more complex operations are encoded at the HSI level. Second, the implementor of the functions to be vertically migrated has a finer level of control than would be possible using constructs above the HSI. Typically, functions that were previously written in the macrocode of a conventional processor are rewritten in microcode, which then becomes part of the hardware.

Hassit and Lyon describe an application of vertical migration in [13] and [12]. This was a vertical migration of selected APL primitives. The primitives were microcoded on an IBM System/370 model 25. Weber describes an implementation of EULER on an IBM System/360 model 30 in [35]. Luque and Ripoll provide a summary and overview of vertical migration in [18]. Pihlgen describes the vertical migration of COBOL primitives in [25]. These are just a few examples of the many papers published in this area.
3.4. REDUCED INSTRUCTION SETS

The last idea about the HSI and DSI that will be discussed is the idea of "reduced instruction set" machines. There is no consensus on what defines a reduced instruction set machine. We will consider in this section only those machines that put the DSI and the HSI at the SCI, since we believe this is critical issue. Examples of these machines are the IBM 801 [26], the Berkeley RISC [23], [24], the Stanford MIPS [14], [15], and the HP Spectrum [2], [19]. Two other machines that have claimed to be reduced instruction set machines, the Ridge 32 [11], and the Pyramid 90X [27] may have lowered the HSI and the DSI somewhat compared with some machines, but they are both still above the SCI.

The unique feature of the reduced instruction set idea is to lower the DSI as well as the HSI. The dynamic microprogramming idea discussed above advocated the lowering of the HSI, but not the DSI. Many papers justifying the 'reduced instruction set' concept seem to ignore the fact that the DSI is lower as well as the HSI. For example, Radin, in [26], makes the following statement with regard to the IBM 801:

"... the benefits claimed of microcode are generally not due to the power of the instructions as much as to their residence in a high-speed control store. This amounts to a hard-ware architect attempting to guess which subroutines, or macros, are most frequently used and assigning high-speed memory to them. ... The 801 CPU gets its instructions from an 'instruction cache' which is managed by least-recently-used information. Thus, all frequently used functions are very likely to be found in this high-speed storage, ..." [26]

Birnbaum and Worley, in [2], p. 41, also make an equivalent remark about the HP Spectrum family. What they fail to make clear is that the DSI has been lowered as well as the HSI. Making the comparison between a control store and an instruction cache isn’t as simple as they might have you believe since one has microinstructions below the DSI and the other has microinstructions above the DSI. These are two very different situations and a comparison is not simple. Besides, some computers cache the control store without lowering the DSI, for example the Burroughs B1726 [21]. The hit ratio of a control store cache depends only on the frequency of individual instructions. The hit ratio of an instruction cache, on the other hand, depends on instruction stream locality, compiler technology, and how dynamic the environment is.

4. CONCLUSIONS

The computer architect is faced with many tradeoffs. In order to appropriately evaluate these tradeoffs, many different tools are needed. In particular, it is necessary to capture the significant features of different designs while ignoring the insignificant ones. We have attempted to aid this process by clarifying the concept of the dynamic/static interface and comparing it to the hardware/software interface.

The interfaces themselves are not new, but we have combined them in a coherent manner and defined a two dimensional space that has each interface as an axis. This space, shown in figure 1, illustrates the usefulness of the HSI/DSI concept. Different approaches to computer design can be put into perspective with other approaches. Of course, this captures only some of the characteristics of a design, but we believe them to be significant ones.

This paper represents research in progress. Most of the ideas presented here are still developing. We welcome feedback and encourage discussion.

ACKNOWLEDGEMENT

We gratefully acknowledge our colleagues, students and faculty, within the Aquarius program for providing the stimulating environment at Berkeley within which we work. We also acknowledge that part of this work was sponsored by the Defense Advanced Research Projects Agency (DoD), Arpa Order No. 4871, monitored by Space and Naval Warfare Systems Command under Contract No. N00039-84-C-0089.

REFERENCES


[31] Rex Rice, "The Chief Architect's Reflections


BIBLIOGRAPHY


