Efficient RAM and control flow in verifiable outsourced computation

Riad S. Wahby*, Srinath Setty†‡, Zuocheng Ren†, Andrew J. Blumberg†, and Michael Walfish*

*New York University
†The University of Texas at Austin
‡now at Microsoft Research

February 10, 2015
Proof-based verifiable computation enables outsourcing

**Goal:** A client wants to outsource a computation

- with strong correctness guarantees, and
- without assumptions about the server’s hardware or how failures might occur.
Proof-based verifiable computation enables outsourcing

Approach: Server’s response includes short proof of correctness.

This solution is based on powerful theoretical tools.

[GMR85, BCC88, BFLS91, ALMSS92, AS92, Kilian92, LFKN92, Shamir92, Micali00, BS05, BGHSV06, IKO07, GKR08]
## Related work in proof-based verification

<table>
<thead>
<tr>
<th>setup costs</th>
<th>applicable computations</th>
<th>general control flow</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>none</strong></td>
<td>Thaler, CMT, TRMP</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[CRYPTO13, ITCS12,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>HotCloud12]</td>
<td></td>
</tr>
<tr>
<td><strong>low</strong></td>
<td>Allspice</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[IEEE S&amp;P13]</td>
<td></td>
</tr>
<tr>
<td><strong>med</strong></td>
<td>Pepper</td>
<td>Ginger</td>
</tr>
<tr>
<td></td>
<td>[NDSS12]</td>
<td>[Security12]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Zaatar</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[Eurosys13]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Pinocchio</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[IEEE S&amp;P13]</td>
</tr>
<tr>
<td><strong>high</strong></td>
<td></td>
<td>Pantry</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[SOSP13]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BCTV, BCGTV</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[Security14,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CRYPTO13]</td>
</tr>
</tbody>
</table>
# Related work in proof-based verification

<table>
<thead>
<tr>
<th>setup costs</th>
<th>applicable computations</th>
<th>general control flow</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>Thaler, CMT, TRMP</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[CRYPTO13, ITCS12, HotCloud12]</td>
<td></td>
</tr>
<tr>
<td>low</td>
<td>Allspice</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[IEEE S&amp;P13]</td>
<td></td>
</tr>
<tr>
<td>med</td>
<td>Pepper</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[NDSS12]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Ginger</td>
<td>Pantry</td>
</tr>
<tr>
<td></td>
<td>[Security12]</td>
<td>[SOSP13]</td>
</tr>
<tr>
<td></td>
<td>Zaatar</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[Eurosys13], Pinocchio</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[IEEE S&amp;P13]</td>
<td></td>
</tr>
<tr>
<td>high</td>
<td>BCTV, BCGTV</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[Security14, CRYPTO13]</td>
<td></td>
</tr>
</tbody>
</table>
### Related work in proof-based verification

<table>
<thead>
<tr>
<th>setup</th>
<th>applicable computations</th>
</tr>
</thead>
<tbody>
<tr>
<td>costs</td>
<td>regular structure</td>
</tr>
<tr>
<td>none</td>
<td>Thaler, CMT, TRMP</td>
</tr>
<tr>
<td></td>
<td>[CRYPTO13, ITCS12, HotCloud12]</td>
</tr>
<tr>
<td>low</td>
<td>Allspice</td>
</tr>
<tr>
<td></td>
<td>[IEEE S&amp;P13]</td>
</tr>
<tr>
<td>med</td>
<td>Pepper</td>
</tr>
<tr>
<td></td>
<td>[NDSS12]</td>
</tr>
<tr>
<td></td>
<td>Ginger</td>
</tr>
<tr>
<td></td>
<td>[Security12]</td>
</tr>
<tr>
<td></td>
<td>Zaatar</td>
</tr>
<tr>
<td></td>
<td>[Eurosys13], Pinocchio</td>
</tr>
<tr>
<td></td>
<td>[IEEE S&amp;P13]</td>
</tr>
<tr>
<td>high</td>
<td>Pantry</td>
</tr>
<tr>
<td></td>
<td>[SOSP13]</td>
</tr>
<tr>
<td></td>
<td>BCTV, BCGTV</td>
</tr>
<tr>
<td></td>
<td>[Security14, CRYPTO13]</td>
</tr>
</tbody>
</table>

**better**
## Related work in proof-based verification

<table>
<thead>
<tr>
<th>setup costs</th>
<th>applicable computations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>regular structure</td>
</tr>
<tr>
<td>none</td>
<td>Thaler, CMT, TRMP</td>
</tr>
<tr>
<td></td>
<td>[CRYPTO13, ITCS12, HotCloud12]</td>
</tr>
<tr>
<td>low</td>
<td>Allspice</td>
</tr>
<tr>
<td></td>
<td>[IEEE S&amp;P13]</td>
</tr>
<tr>
<td>med</td>
<td>Pepper</td>
</tr>
<tr>
<td></td>
<td>[NDSS12]</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>high</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Buffet* (this work)
Verifiable computation still faces challenges

Tension between expressiveness and efficiency

Large (amortized) setup costs for the client; massive server overhead

Buffet
(this work)

Substantially mitigated

Not addressed
The rest of this talk

1. Background: the proof-based verification framework

2. Buffet: dynamic control flow in arithmetic circuits

3. Experimental results
The rest of this talk

1. Background: the proof-based verification framework

2. Buffet: dynamic control flow in arithmetic circuits

3. Experimental results
Verifiable computation overview: common machinery

Buffet and its predecessors share a common framework.
Verifiable computation overview: common machinery

Buffet and its predecessors share a common framework.
Verifiable computation overview: common machinery

Buffet and its predecessors share a common framework.

```
front-end
arithmetic circuit ⇔ program

program
(subset of C)

arithmetic circuit

theoretical tools
(e.g., PCPs)

back-end

client executable

server executable

client executable

server executable

inputs

outputs, proof
```
Verifiable computation overview: common machinery

Buffet and its predecessors share a common framework.

**front-end**
- arithmetic circuit
- program (subset of C)

**back-end**
- valid proof $\implies$ execution follows arithmetic circuit, respects inputs
- theoretical tools (e.g., PCPs)
- client executable
- server executable

**client executable**
- inputs
- outputs, proof

**server executable**
- inputs
- outputs, proof
Verifiable computation overview: common machinery

Buffet and its predecessors share a common framework.

 Costs scale with arithmetic circuit size. So:

How can Buffet’s front-end efficiently represent general-purpose C programs in arithmetic circuits?
These compilers handle a subset of C:

1. Assignment: allocate a fresh wire for each assignment.

\[
\begin{align*}
i &= i + 1; \quad \Rightarrow \quad i_1 &= i_0 + 1;
\end{align*}
\]
Compiling programs to circuits in Pantry [SOSP13]
(and Zaatar [Eurosys13] and Pinocchio [IEEE S&P13])

These compilers handle a subset of C:

1. Assignment: allocate a fresh wire for each assignment.
2. Conditionals: execute both branches and select desired result.

```c
if (i > 5)
    i = i + 1;
else
    i = i * 2;
```

⇒

```c
i1 = i0 + 1;
i2 = i0 * 2;
i3 = (i0 > 5) ?
    i1 : i2;
```
These compilers handle a subset of C:

1. Assignment: allocate a fresh wire for each assignment.
2. Conditionals: execute both branches and select desired result.
3. Loops: unroll at compile time. Loop bounds must be static.

```c
i=0;
for (j=0; j<10; j++) {
    i++;
}
```

```c
i = 0;
i0 = i + 1; // j == 0
i1 = i0 + 1; // j == 1
...i9 = i8 + 1; // j == 9
```
Compiling programs to circuits in Pantry [SOSP13](and Zaatar [Eurosys13] and Pinocchio [IEEE S&P13])

These compilers handle a subset of C:

1. Assignment: allocate a fresh wire for each assignment.
2. Conditionals: execute both branches and select desired result.
3. Loops: unroll at compile time. Loop bounds must be static.
4. Arithmetic, inequalities, and logical operations are supported.
These compilers handle a subset of C:

1. Assignment: allocate a fresh wire for each assignment.
2. Conditionals: execute both branches and select desired result.
3. Loops: unroll at compile time. Loop bounds must be static.
4. Arithmetic, inequalities, and logical operations are supported.

Buffet’s key challenge: how can we support general C programs with arbitrary control flow, including break, continue, and data dependent looping?
These compilers handle a subset of C:

1. **Assignment**: allocate a fresh wire for each assignment.
2. **Conditionals**: execute both branches and select desired result.
3. **Loops**: unroll at compile time. Loop bounds must be **static**.
4. **Arithmetic, inequalities, and logical operations** are supported.

Buffet’s **key challenge**: how can we support *general* C programs with arbitrary control flow, including **break**, **continue**, and data dependent looping?

Buffet also adapts and refines a previous approach to verified RAM [BCGT12, BCGTV13, BCTV14] (see paper).
The rest of this talk

1. Background: the proof-based verification framework

2. Buffet: dynamic control flow in arithmetic circuits

3. Experimental results
Compiling nested loops

In a loop nest, inner loop unrolls into every iteration of outer loop.

\[
\begin{align*}
\text{i} &= 0; \\
\text{for (j=0; j<10; j++)} \{ \\
&\quad \text{i++;} \\
&\quad \text{for (k=0; k<2; k++)} \{ \\
&\quad\quad \text{i=i*2;} \\
&\quad\} \\
&\} \\
\end{align*}
\]

\[
\begin{align*}
\text{i} &= 0; \\
\text{i0} &= \text{i+1}; & // j == 0 \\
\text{i1} &= \text{i0*2}; & // k == 0 \\
\text{i2} &= \text{i1*2}; & // k == 1 \\
\text{i3} &= \text{i2+1}; & // j == 1 \\
\text{i4} &= \text{i3*2}; & // k == 0 \\
\text{i5} &= \text{i4*2}; & // k == 1 \\
\cdots
\end{align*}
\]
Compiling nested loops with data dependent bounds

Consider a decoder for a run-length encoded string with output size OUTLENGTH:

“a5b2” ⇒ “aaaaabb”

```c
i = j = 0;
while (j < OUTLENGTH) {
    inchar = input[i++];
    length = input[i++];

    do {
        output[j++] = inchar;
        length--;
    } while (length > 0);
}
```
Compiling nested loops with data dependent bounds

Consider a decoder for a run-length encoded string with output size \text{OUTLENGTH}:

“a5b2” ⇒ “aaaaabb”

\[
i = j = 0;
\]

\[
\text{while (j < OUTLENGTH) \{} \]
\[
\text{inchar = input[i++]}; \]
\[
\text{length = input[i++]}; \]
\[
\text{do \{} \]
\[
\text{output[j++] = inchar}; \]
\[
\text{length--}; \]
\[
\text{\} while (length > 0); \]
\[
\text{\} \}
\]

1. Read (inchar, length) pair.
Compiling nested loops with data dependent bounds

Consider a decoder for a run-length encoded string with output size $\text{OUTLENGTH}$:

“a5b2” $\Rightarrow$ “aaaaabb”

```
i = j = 0;
while (j < OUTLENGTH) {
    inchar = input[i++];
    length = input[i++];

    do {
        output[j++] = inchar;
        length--;
    } while (length > 0);
}
```

1. Read (inchar, length) pair.
2. Emit inchar, length times.
Compiling nested loops with data dependent bounds

Consider a decoder for a run-length encoded string with output size \texttt{OUTLENGTH}:

“a5b2” ⇒ “aaaaabb”

```c
i = j = 0;
while (j < \texttt{OUTLENGTH}) {
    inchar = \texttt{input[i++]};
    length = \texttt{input[i++]};

    do {          /* bound= ???        */
        output[j++] = inchar;
        length--;
    } while (length > 0);
}
```

At one extreme, a single character’s run length could be \texttt{OUTLENGTH}. so this must be the inner bound.
Compiling nested loops with data dependent bounds

Consider a decoder for a run-length encoded string with output size OUTLENGTH:

“a5b2” ⇒ “aaaaaabb”

```c
i = j = 0;
while (j < OUTLENGTH) {
    inchar = input[i++];
    length = input[i++];

    do {
        /* bound=OUTLENGTH */
        output[j++] = inchar;
        length--;
    } while (length > 0);
}
```
Compiling nested loops with data dependent bounds

Consider a decoder for a run-length encoded string with output size OUTLENGTH:

“a5b2” ⇒ “aaaaabb”

```c
i = j = 0;
while (j < OUTLENGTH) { /* bound= ??? */
inchar = input[i++];
length = input[i++];

    do { /* bound=OUTLENGTH */
        output[j++] = inchar;
        length--;
    } while (length > 0);
}
```

At the other extreme, every character’s run length could be 1, and the outer loop would iterate OUTLENGTH times.
Compiling nested loops with data dependent bounds

Consider a decoder for a run-length encoded string with output size OUTLENGTH:

“a5b2” ⇒ “aaaaabb”

```
i = j = 0;
while (j < OUTLENGTH) {  /* bound=OUTLENGTH */
    inchar = input[i++];
    length = input[i++];

    do {
        output[j++] = inchar;
        length--;
    } while (length > 0);
}
```

But: this code executes OUTLENGTH^2 inner loop iterations, and the resulting arithmetic circuit is quadratic in OUTLENGTH.
We can’t eliminate unrolling. What about nesting?
We can’t eliminate unrolling. What about nesting?

Consider:

1. Loop nests are equivalent to finite state machines.
2. Arithmetic circuits can efficiently represent FSMs.
We can’t eliminate unrolling. What about nesting?

Consider:

1. Loop nests are equivalent to finite state machines.
2. Arithmetic circuits can efficiently represent FSMs.

Idea: transform loop nests into FSMs.
FSM Transformation: step 1

We can build a control flow graph for the RLE decoder:

```c
i = j = 0;
while (j < OUTLENGTH) {
    inchar = input[i++];
    length = input[i++];

    do {
        output[j++] = inchar;
        length--;
    } while (length > 0);
}
```

1. Identify vertices: straight line code segments.
2. Identify edges: control flow between segments.

1 transitions to 2 unconditionally.
2 self-transitions when `LENGTH > 0`.
2 transitions to 1 when `LENGTH <= 0`.
FSM Transformation: step 1

We can build a control flow graph for the RLE decoder:

```c
i = j = 0;
while (j < OUTLENGTH) {
    inchar = input[i++];
    length = input[i++];
    do {
        output[j++] = inchar;
        length--;
    } while (length > 0);
}
```

1. Identify vertices: straight line code segments.
2. Identify edges: control flow between segments.
   1 transitions to 2 unconditionally.
   2 self-transitions when \( \text{LENGTH} > 0 \).
   2 transitions to 1 when \( \text{LENGTH} \leq 0 \).
FSM Transformation: step 1

We can build a control flow graph for the RLE decoder:

```c
i = j = 0;
while (j < OUTLENGTH) {
    inchar = input[i++];
    length = input[i++];
    do {
        output[j++] = inchar;
        length--;
    } while (length > 0);
}
```

1. Identify vertices: straight line code segments.
2. Identify edges: control flow between segments.
   1 transitions to 2 unconditionally.
FSM Transformation: step 1

We can build a control flow graph for the RLE decoder:

```java
i = j = 0;
while (j < OUTLENGTH) {
    inchar = input[i++];
    length = input[i++];

    do {
        output[j++] = inchar;
        length--;
    } while (length > 0);
}
```

1. Identify vertices: straight line code segments.
2. Identify edges: control flow between segments.
   1 transitions to 2 unconditionally.
   2 self-transitions when length > 0.
We can build a control flow graph for the RLE decoder:

```
i = j = 0;
while (j < OUTLENGTH) {
  inchar = input[i++];
  length = input[i++];
  do {
    output[j++] = inchar;
    length--;
  } while (length > 0);
}
```

1. Identify vertices: straight line code segments.
2. Identify edges: control flow between segments.
   1 transitions to 2 unconditionally.
   2 self-transitions when length > 0.
   2 transitions to 1 when length <= 0.
FSM Transformation: step 2

From the control flow graph

1

length <= 0

2

length > 0
FSM Transformation: step 2

From the control flow graph, we can build a state machine.

```c
i = j = 0;
state = 1;
while (j < OUTLENGTH) {
    if (state == 1) {
        inchar = input[i++];
        length = input[i++];
        state = 2;
    }
    if (state == 2) {
        output[j++] = inchar;
        length--;
        if (length <= 0) {
            state = 1;
        }
    }
}
```
From the control flow graph, we can build a state machine.

```c
i = j = 0;
while (j < OUTLENGTH) {
    inchar = input[i++];
    length = input[i++];
    do {
        output[j++] = inchar;
        length--;
    } while (length > 0);
}
```

```c
i = j = 0;
state = 1;
while (j < OUTLENGTH) {
    if (state == 1) {
        inchar = input[i++];
        length = input[i++];
    }
    if (state == 2) {
        output[j++] = inchar;
        length--;
    }
    if (length <= 0) {
        state = 1;
    }
}
```
Buffet’s FSM transformation: *loop flattening*

Buffet’s transformation extends *loop flattening* [Ghuloum & Fisher, PPOPP95] with support for arbitrary loops, break, and continue.
Buffet’s FSM transformation: *loop flattening*

Buffet’s transformation extends *loop flattening* [Ghuloum & Fisher, PPOPP95] with support for arbitrary loops, break, and continue.

Caveats:

- Programmer must tell Buffet # of steps to unroll the FSM.
- No goto in Buffet’s implementation (yet).
- No “program memory” $\Rightarrow$ no function pointers.
What if we unrolled a whole CPU? [BCTV, Security14]

The state variable in the FSM is like a coarse program counter. What if we just had a program counter, registers, etc?
What if we unrolled a whole CPU? [BCTV, Security14]

The state variable in the FSM is like a coarse program counter. What if we just had a program counter, registers, etc?

This is the approach of BCTV: Represent a CPU transition

```
fetch-decode-
execute
CPU state:
  pc, regs, ...
```
What if we unrolled a whole CPU? [BCTV, Security14]

The state variable in the FSM is like a coarse program counter. What if we just had a program counter, registers, etc?

This is the approach of BCTV:
Represent a CPU transition, and unroll it.

- fetch-decode-execute step 1
  - CPU state: pc, regs, ...

- fetch-decode-execute step 2
  - CPU state: pc, regs, ...

- ...

- fetch-decode-execute step T
  - CPU state: pc, regs, ...
What if we unrolled a whole CPU? [BCTV, Security14]

The state variable in the FSM is like a coarse program counter. What if we just had a program counter, registers, etc?

This is the approach of BCTV:
Represent a CPU transition, and unroll it.

fetch-decode-execute step 1

CPU state: pc, regs, ...

fetch-decode-execute step 2

CPU state: pc, regs, ...

⋯

fetch-decode-execute step T

CPU state: pc, regs, ...

BCTV supports all of C, but like other systems requires bounded execution (programmer chooses # of CPU steps).
What if we unrolled a whole CPU? [BCTV, Security14]

The state variable in the FSM is like a coarse program counter. What if we just had a program counter, registers, etc?

This is the approach of BCTV:
Represent a CPU transition, and unroll it.

BCTV supports all of C, but like other systems requires bounded execution (programmer chooses # of CPU steps).

But: BCTV pays the cost of an entire CPU for each program step.
The rest of this talk

1. Background: the proof-based verification framework

2. Buffet: dynamic control flow in arithmetic circuits

3. Experimental results
Evaluation questions

Front-end

- Arithmetic circuit
- Program (subset of C)

Back-end

- Valid proof
- Execution follows arithmetic circuit, respects inputs
- Theoretical tools (e.g., PCPs)
- Client executable
- Server executable
Evaluation questions

Using the same back-end for Pantry, BCTV, and Buffet, how do the front-ends compare?

1. For a fixed arithmetic circuit size, what is the maximum computation size each system can handle?
Evaluation questions

Using the same back-end for Pantry, BCTV, and Buffet, how do the front-ends compare?

1. For a fixed arithmetic circuit size, what is the maximum computation size each system can handle?

2. For a fixed computation size, what is the server’s cost under each system?
Implementation

Buffet front-end: builds on Pantry [Braun et al., SOSP13].
FSM transform: source-to-source compiler built on top of clang.
Implementation

Buffet front-end: builds on Pantry [Braun et al., SOSP13].
FSM transform: source-to-source compiler built on top of clang.

For evaluation, we reimplemented the BCTV system, including

- a toolchain for the simulated CPU in Java and C
- a CPU simulator in C, compiled using Pantry

Our implementation’s performance is within 15% of BCTV.
Implementation

Buffet front-end: builds on Pantry [Braun et al., SOSP13].
FSM transform: source-to-source compiler built on top of clang.

For evaluation, we reimplemented the BCTV system, including
- a toolchain for the simulated CPU in Java and C
- a CPU simulator in C, compiled using Pantry

Our implementation’s performance is within 15% of BCTV.

We use the Pinocchio back-end [Parno et al., IEEE S&P13].
(Highly optimized implementation from BCTV [Security14].)
Implementation


For evaluation, we reimplemented the BCTV system, including
- a toolchain for the simulated CPU in Java and C
- a CPU simulator in C, compiled using Pantry

Our implementation’s performance is within 15% of BCTV.

We use the Pinocchio back-end [Parno et al., IEEE S&P13]. (Highly optimized implementation from BCTV [Security14].)

Evaluation platform:
- Texas Advanced Computing Center (TACC), Stampede cluster
- Linux machines with Intel Xeon E5-2680, 32 GB of RAM
What is the maximum computation size for each system?

For an arithmetic circuit of \( \approx 10^7 \) gates, we have:

<table>
<thead>
<tr>
<th></th>
<th>Pantry</th>
<th>BCTV</th>
<th>Buffet</th>
</tr>
</thead>
<tbody>
<tr>
<td>matrix multiplication ( m \times m )</td>
<td>( m = 215 )</td>
<td>( m = 7 )</td>
<td>( m = 215 )</td>
</tr>
<tr>
<td>merge sort ( k ) elements</td>
<td>( k = 8 )</td>
<td>( k = 32 )</td>
<td>( k = 512 )</td>
</tr>
<tr>
<td>Knuth-Morris-Pratt search</td>
<td>( n = 4, \ell = 8 )</td>
<td>( n = 16, \ell = 160 )</td>
<td>( n = 256, \ell = 2900 )</td>
</tr>
</tbody>
</table>
What is the maximum computation size for each system?

For an arithmetic circuit of \( \approx 10^7 \) gates, we have:

<table>
<thead>
<tr>
<th></th>
<th>Pantry</th>
<th>BCTV</th>
<th>Buffet</th>
</tr>
</thead>
<tbody>
<tr>
<td>matrix multiplication</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( m \times m )</td>
<td>( m = 215 )</td>
<td>( m = 7 )</td>
<td>( m = 215 )</td>
</tr>
<tr>
<td>merge sort</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( k ) elements</td>
<td>( k = 8 )</td>
<td>( k = 32 )</td>
<td>( k = 512 )</td>
</tr>
<tr>
<td>Knuth-Morris-Pratt search</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>needle length = ( n )</td>
<td>( n = 4, )</td>
<td>( n = 16, )</td>
<td>( n = 256, )</td>
</tr>
<tr>
<td>haystack length = ( \ell )</td>
<td>( \ell = 8 )</td>
<td>( \ell = 160 )</td>
<td>( \ell = 2900 )</td>
</tr>
</tbody>
</table>
What is the maximum computation size for each system?

For an arithmetic circuit of \( \approx 10^7 \) gates, we have:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Pantry</th>
<th>BCTV</th>
<th>Buffet</th>
</tr>
</thead>
<tbody>
<tr>
<td>matrix multiplication ( m \times m )</td>
<td>( m = 215 )</td>
<td>( m = 7 )</td>
<td>( m = 215 )</td>
</tr>
<tr>
<td>merge sort ( k ) elements</td>
<td>( k = 8 )</td>
<td>( k = 32 )</td>
<td>( k = 512 )</td>
</tr>
<tr>
<td>Knuth-Morris-Pratt search</td>
<td>( n = 4, )  ( n = 16, )  ( n = 256, )</td>
<td>( \ell = 8 )  ( \ell = 160 )</td>
<td>( \ell = 2900 )</td>
</tr>
</tbody>
</table>
What is the maximum computation size for each system?

For an arithmetic circuit of $\approx 10^7$ gates, we have:

<table>
<thead>
<tr>
<th></th>
<th>Pantry</th>
<th>BCTV</th>
<th>Buffet</th>
</tr>
</thead>
<tbody>
<tr>
<td>matrix multiplication</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$m \times m$</td>
<td>$m = 215$</td>
<td>$m = 7$</td>
<td>$m = 215$</td>
</tr>
<tr>
<td>merge sort</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$k$ elements</td>
<td>$k = 8$</td>
<td>$k = 32$</td>
<td>$k = 512$</td>
</tr>
<tr>
<td>Knuth-Morris-Pratt search</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>needle length = $n$</td>
<td>$n = 4$, $n = 16$, $n = 256$</td>
<td>$n = 16$, $n = 256$, $n = 256$</td>
<td>$n = 16$, $n = 256$, $n = 256$</td>
</tr>
<tr>
<td>haystack length = $\ell$</td>
<td>$\ell = 8$</td>
<td>$\ell = 160$</td>
<td>$\ell = 2900$</td>
</tr>
</tbody>
</table>
What is the maximum computation size for each system?

For an arithmetic circuit of \( \approx 10^7 \) gates, we have:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Pantry</th>
<th>BCTV</th>
<th>Buffet</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Matrix multiplication</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( m \times m )</td>
<td>( m = 215 )</td>
<td>( m = 7 )</td>
<td>( m = 215 )</td>
</tr>
<tr>
<td><strong>Merge sort</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( k ) elements</td>
<td>( k = 8 )</td>
<td>( k = 32 )</td>
<td>( k = 512 )</td>
</tr>
<tr>
<td><strong>Knuth-Morris-Pratt search</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Needle length = ( n )</td>
<td>( n = 4, )</td>
<td>( n = 16, )</td>
<td>( n = 256, )</td>
</tr>
<tr>
<td>Haystack length = ( \ell )</td>
<td>( \ell = 8 )</td>
<td>( \ell = 160 )</td>
<td>( \ell = 2900 )</td>
</tr>
</tbody>
</table>

These data establish ground truth. For apples-to-apples front-end comparison, we now extrapolate to Buffet’s computation sizes.
What is the server’s cost for each system?

Extrapolated server execution time, normalized to Buffet

matrix multiplication
\[ m = 215 \]

merge sort
\[ k = 512 \]

Knuth-Morris-Pratt search
\[ n = 256, \; \ell = 290 \]
But we still have a long way to go!

Extrapolated server execution time, normalized to native execution

- Matrix multiplication: $m=215$
- Merge sort: $k=512$
- Knuth-Morris-Pratt search: $n=256$, $\ell=290$
Recap

Buffet combines the best aspects of Pantry and BCTV.

- Straight line computations are very efficient.
- Buffet charges the programmer only for what is used.
- General looping is transformed into FSM, efficiently compiled.
- RAM interactions are efficient (see paper).

Buffet improves on Pantry and BCTV by 1–4 orders of magnitude.
Recap

Buffet combines the best aspects of Pantry and BCTV.

+ Straight line computations are very efficient.
+ Buffet charges the programmer only for what is used.
+ General looping is transformed into FSM, efficiently compiled.
+ RAM interactions are efficient (see paper).

Buffet improves on Pantry and BCTV by 1–4 orders of magnitude.

Buffet still has limitations:

- No support for goto or function pointers.
- Like all systems in the area, server overheads are still massive.
Recap

Buffet combines the best aspects of Pantry and BCTV.
+ Straight line computations are very efficient.
+ Buffet charges the programmer only for what is used.
+ General looping is transformed into FSM, efficiently compiled.
+ RAM interactions are efficient (see paper).

Buffet improves on Pantry and BCTV by 1–4 orders of magnitude.

Buffet still has limitations:
  – No support for goto or function pointers.
  – Like all systems in the area, server overheads are still massive.

http://www.pepper-project.org/